Golden System Reference Design and Demo User Guide v2.0 for Avant-E Devices

Lattice Propel 2025.1.1
Lattice Radiant 2025.1.1

Reference Design

FPGA-RD-02324-1.2

July 2026

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 REDUNDANCIES, 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

Abbreviations in This Document

A list of abbreviations used in this document.

Abbreviation	Definition
AHBL	Advanced High-performance Bus-Lite
AXI	Advanced eXtensible Interface
APB	Advanced Peripheral Bus
API	Application Programming Interface
AXI4-Lite	Advanced eXtensible Interface-Lite
GPIO	General Purpose Input/Output
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DDR	Double Data Rate
FIFO	First-In-First-Out
FMC	FPGA Mezzanine Card
GHRD	Golden Hardware Reference Design
GSRD	Golden System Reference Design
ICMP	Internet Control Message Protocol
ISR	Interrupt Service Routines
LMMI	Lattice Memory Mapped Interface
LPDDR4	Low Power Double Data Rate Generation 4
lwIP	lightweight Internet Protocol
OPN	Ordering Part Number
PLL	Phase-Locked Loop
QSPI	Quad Serial Peripheral Interface
RGMII	Reduced Gigabit Media Independent Interface
RISC-V	Reduced Instruction Set Computer-V
RTL	Register Transfer Level
SGDMA	Scatter-Gather Direct Memory Access
SoC	System on Chip
TSE MAC	Tri-Speed Ethernet Media Access Controller
UART	Universal Asynchronous Receiver-Transmitter

1. Introduction

The Lattice FPGA-based Golden System Reference Design (GSRD) presented herein is aimed at providing a versatile and efficient platform for embedded applications requiring high-performance computing, memory access, data transfer capabilities, and network communication. By integrating various components onto a single FPGA chip, this design offers flexibility, scalability, and cost-effectiveness for a wide range of applications.

The Lattice Golden Hardware Reference Design (GHRD) is a System-on-Chip (SoC) that can be used as a baseline design to create FPGA applications as per the user requirements. It is a RISC-V based design that interacts with various Lattice Soft-IPs and peripherals such as GPIO, UART, Tri-Speed Ethernet (TSE), Octal SPI Controller, Scatter-Gather DMA (SGDMA) Controller and DDR Memory Controller. All these building blocks are connected through industry standard protocols such as AXI-MM, AXI-Steam for data transfers and APB for control.

The GSRD is a comprehensive embedded system which incorporates drivers and relevant firmware needed to operate various design components. FreeRTOS and bootloader are built on RISC-V RX CPU. The primary function of bootloader is to initialize the hardware blocks in the design using the respective IP drivers and ensure the integrity of the application software by performing CRC check.

Hardware and software are integrated together to establish a complete system for which relevant binaries and executables are generated by Lattice Software tools such as Propel SDK, Propel Builder and Radiant to program the FPGA Hardware.

As a part of multi-boot demonstration, the executables folder comprises compatible binaries and executable images, that is FPGA bitstream (.bit) and software binary (.bin) for both Primary and Golden GSRD projects. The only difference is that the Primary bitstream contains the code for multi-boot enablement and Golden bitstream does not enable multi-boot.

GSRD for the Lattice Avant™-E device is developed and tested with the Lattice Propel™ and Lattice Radiant™ software versions as indicated in Table 1.1.

1.1. Quick Facts

Table 1.1. Summary of the System

SoC Requirements	Supported FPGA Family	Avant-E
	SoC Version	2.0
FPGA Device(s)	Targeted Board	Avant-E70 Evaluation Board Rev D OPN: LAV-E70-EVN-ES2
	Targeted Device	LAV-AT-E70-3LFG1156I
	Supported User Interface	AXI-MM, AXI-Stream, APB
Design Tool Support	Lattice Implementation	Lattice Propel Software 2025.1.1 Lattice Radiant Software 2025.1.1
	Synthesis	Synopsys® Synplify Pro®

1.2. Features

The key features of the system include:

- FPGA device supported in this document is Avant-E70
- RISC-V RX CPU, SGDMA Controller, TSE IP, DDR Memory Controller, and Octal SPI Controller over AXI4 Interface
- 16 Gbit of LPDDR4 cacheable memory with 32-bit DDR data width at 1600 Mbps data rate
- 1 Gbps Ethernet throughput through RGMII support at 125 MHz
- Low-speed peripherals like GPIO and UART
- Primary and golden bootloader and FreeRTOS application software
- Application software CRC check by function implemented in RISC-V RX bootloader code
- FPGA bitstream CRC check done by FPGA Configuration Engine
- FreeRTOS application software is run on external LPDDR4 Memory Device
    - Manual and Automatic Dual-Boot capability

1.3. Comparison to SoC Version 1.0

The following are the key changes from SoC version 1.0:

- GHRD based on Avant-E70 ES2 (LAV-AT-E70-3LFG1156I)
- Enhanced architecture for performance and area.
- Centralized interconnect for flow control.
- Multiple clock domains and IP specific clocking scheme for performance.
- Enabled LPDDR4 device in cacheable region and data width converter in DDR Memory Controller.
- Replaced QSPI Flash Controller with Octal SPI Controller.
- Enabled RISC-V watchdog module.
- Updated Radiant, Propel, and soft IPs for 2025.1.1
- Added support for lightweight IP (lwIP) software stack.

1.4. Naming Conventions

1.4.1. Nomenclature

The following are the nomenclature used in this document:

- Boot Up – Process of starting the RISC-V RX CPU, loading the FreeRTOS application software from external SPI Flash into the LPDDR4 memory and executing the application.
- Bootloader – Code that initializes and configures various peripherals and loads the FreeRTOS application software into LPDDR4 memory. It also checks for the CRC of the copied application and decides whether to execute the application software or load the next best hardware bitstream and the corresponding software.
- FreeRTOS application software– Loaded into LPDDR4 memory and executed by RISC-V CPU at the end of boot up process.
- SPI Flash – Non-volatile external memory that stores the bitstream, FreeRTOS application software and multi-boot MCS bitstream.

1.4.2. Signal Names

Signal names that end with:

- _n are active low (asserted when value if logic 0)
- _i are input signals.
- _o are output signals.
- _io are bi-directional input/output signals.

1.5. Prerequisites

The following sections show the software and hardware requirements to run the demonstration and compiling the reference design.

1.5.1. Lattice Software Tools Requirements

- Lattice Propel 2025.1.1 Package – contains both Lattice Propel SDK and Lattice Propel Builder
  - Download here: Lattice Propel Design Environment
- Lattice Radiant 2025.1.1 Package – contains IP Packager, Radiant Software, QuestaSim, and Programmer
  - Download here: Lattice Radiant Software
- Lattice Propel 2025.1.1 Patch for Avant-E Golden System Reference Design
  - Download here: Downloadable Software tab in the GHRD/GSRD Reference Design

1.5.2. Hardware Requirements

Table 1.2 describes the hardware needed to run the GSRD.

Table 1.2. List of Hardware Required by GSRD

Sr No	Hardware Requirements	Quantity	Comment
1	Lattice Avant-E70 ES2 Evaluation Board OPN: LAV-E70-EVN-ES2	1	Avant-E Evaluation Board web page
2	Mini USB Type-A UART cable for programming bitstream, firmware and proper terminal prints	1	Included in Avant-E70 ES2 Board Evaluation Kit
3	Ethernet FMC daughter card for connection over RGMII interface	1	Not included in Avant-E70 ES2 Board Evaluation Kit. To purchase the FMC daughter card, go to the Ethernet FMC website and select part number **OP031-1V8**.
4	Cat6 RJ45 Ethernet cable to connect Avant-E70 ES2 board to the Host PC through Ethernet FMC daughter card	1	Not included in Avant-E70 ES2 Board Evaluation Kit.
5	12 V power adapter for board power	1	Included in Avant-E70 ES2 Board Evaluation Kit

2. Functional Description

The GSRD/GHRD SoC architecture comprises of a RISC-V RX CPU, DDR Memory Controller, SGDMA Controller, Octal SPI Controller, and 1 Gbps TSE IP, interconnected through a combination of high-speed and low-speed bus fabrics such as AXI4 Interconnect and APB Interconnect. This architecture enables seamless communication and data exchange between the components, facilitating efficient operation and system performance.

2.1. GHRD System Architecture Overview

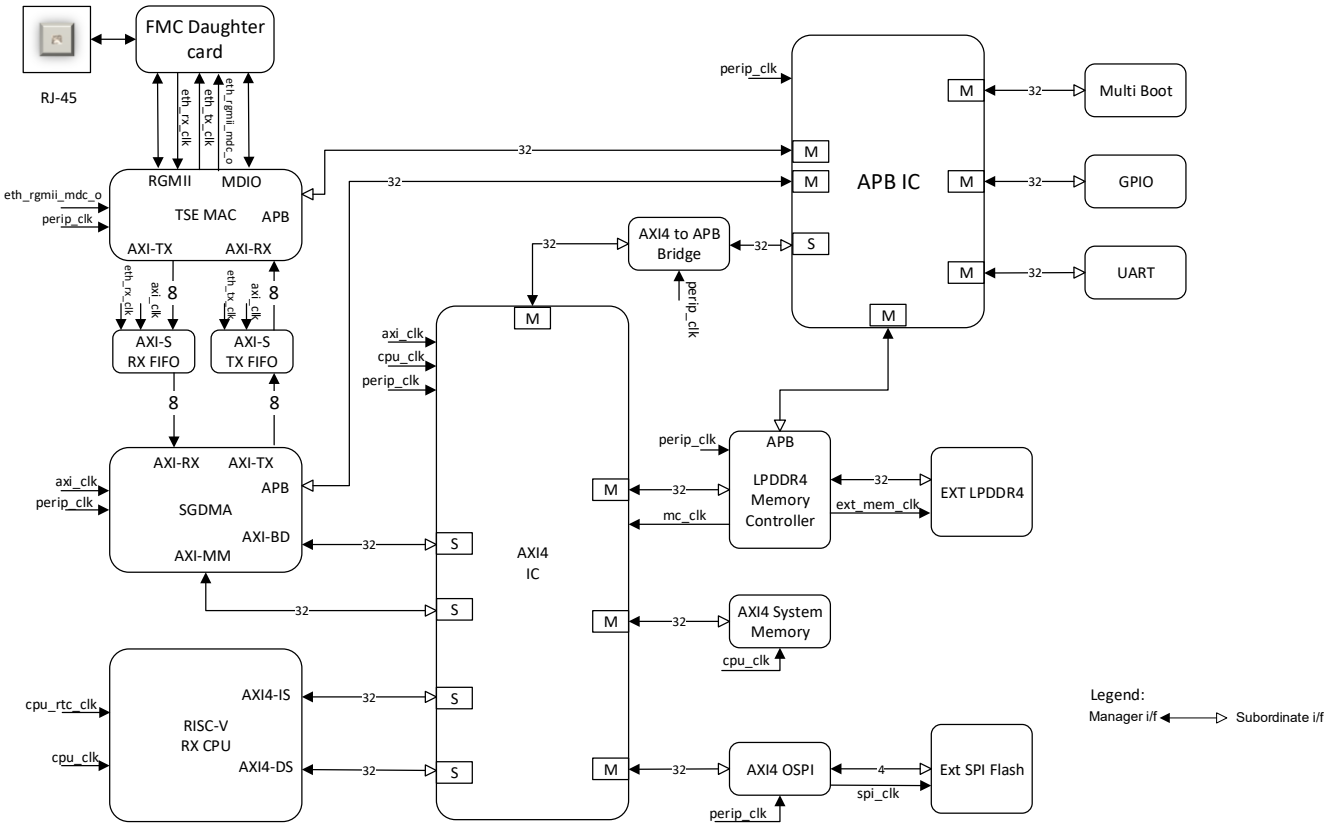


Figure 2.1. GHRD System Architecture

The design includes the following components:

Table 2.1. IP Versions

Soft IP	IP Version
RISC-V RX CPU	2.7.0
DDR Memory Controller	2.6.1
System Memory	2.4.0
Octal SPI Controller	1.3.0
General Purpose I/O	1.8.0
UART	1.4.0
AXI4 Interconnect	2.2.0
APB Interconnect	1.3.0
AXI4 to APB Bridge	1.4.0
Tri-Speed Ethernet	2.1.0
SGDMA Controller	2.5.0

Soft IP	IP Version
Multi-Boot Configuration	1.0.0
PLL	2.6.1
Reset Modules	1.0.0
AXIS FIFO IP	1.0.0

Each component in the block diagram is instantiated using the IP in Propel Builder. The IP features and parameters are described in the IP Configurations section.

The signals on each interface are described in the Signal Description section.

2.2. GHRD Clocking Overview

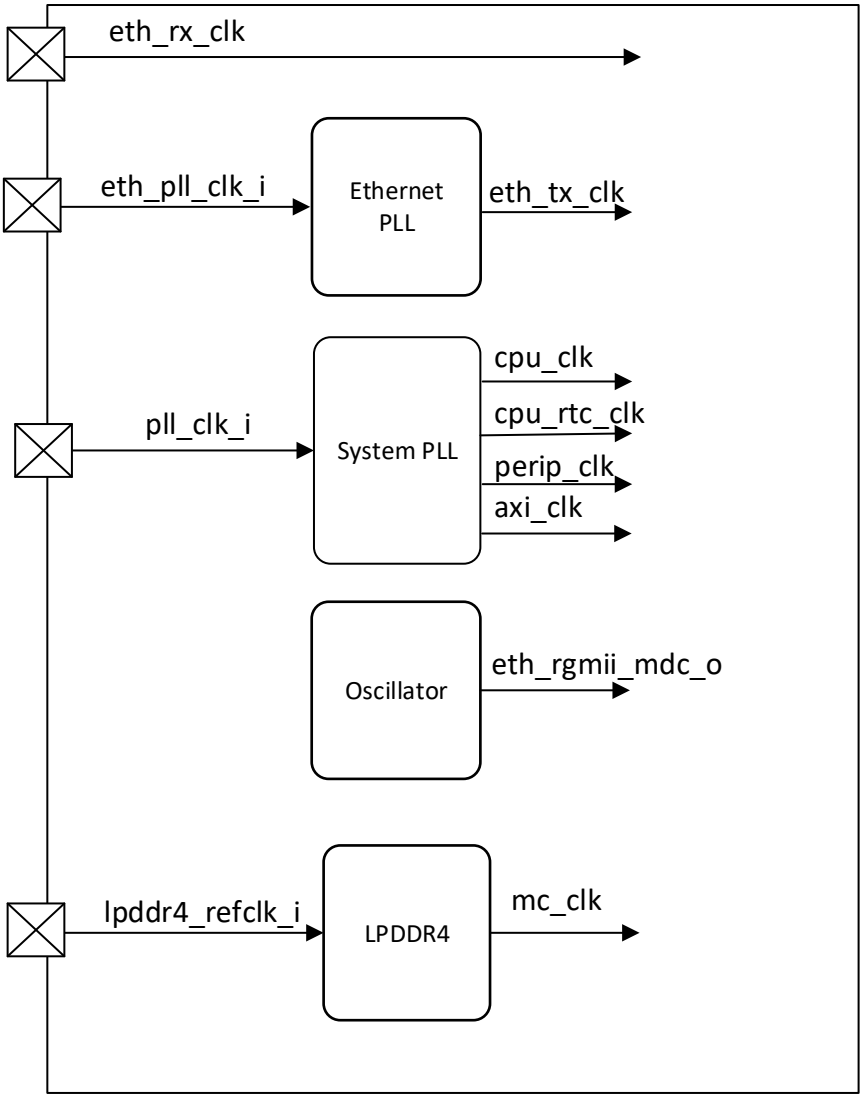


Figure 2.2. Clocking Overview

Table 2.2 describes the reference design clock scheme.

Table 2.2. GHRD Clocking Overview

Sr No	Clock Name	Clock Freq	Clock Source	Destination
1	lpddr4_refclk_i	100 MHz	Onboard oscillator Y1	DDR Memory Controller IP PLL
2	system_pll_clk_i	125 MHz	Onboard oscillator X3	System PLL
3	eth_pll_clk_i	125 MHz	Oscillator on FMC daughter card	Ethernet PLL
4	cpu_clk	200 MHz	system_pll[CLKOP]	RISC-V RX CPU system clock, System Memory, AXI IC S0, S3 and M3 ports
5	perip_clk	100 MHz	system_pll[CLKOS]	AXI IC M0, M1 Ports, TSE IP, SGDMA Controller, UART, GPIO and Multi-boot IP
6	axi_clk	200 MHz	system_pll[CLKOS3]	AXI IC clock, SGDMA Controller
7	eth_tx_clk	125 MHz	eth_pll[CLKOP]	External CDC TX FIFO and TX path of TSE
8	eth_rx_clk	125 MHz	Ethernet RGMII PHY on FMC daughter card	External CDC RX FIFO and RX path of TSE
9	cpu_rtc_clk	16.384 MHz	system_pll[CLKOS2]	RISC-V RX CPU RTC clock
10	mc_clk	200 MHz	DDR Memory Controller	DDR Memory Controller and AXI4 IC (interconnect) M2 port
11	eth_rgmii_mdc_o	10 MHz	On chip oscillator	TSE IP, RGMII PHY

2.3. GHRD Reset Scheme Overview

There are two resets in the entire design:

- External Asynchronous Reset which is controlled by a push button
- Synchronous Reset for entire system is generated from RISC-V

Table 2.3. Reset Scheme

Reset Signal	Source	Destination	Description
system_rstn_i	On-board push button SW1	PLL reset input pin	Reset the PLL and system reset
cpu_rstn_i	Reset sync output port	RISC-V CPU reset input	Reset pin of CPU
system_resetn_o	RISC-V CPU output	All components in system	RISC-V CPU output reset pin, provides reset to all components in the design. This also triggers the reset during CPU OCD debugging mode.
sys_rstn_o	CPU output reset pin	AXI IC S0, S3, AXI IC M3 and System Memory reset input pin	AXI IC CPU subordinates, System Memory, AXI IC M3
perip_rstn_o	CPU output reset pin	AXI IC M0, M1 and all peripheral IPs reset input pin	Octal SPI controller, APB IC, AXI4 to APB bridge, GPIO, UART, SGDMA Controller, TSE IP, DDR Memory Controller and Multi-boot IP

Reset Signal	Source	Destination	Description
axi_rstn_o	CPU output reset pin	AXI IC S1, S2 and SGDMA Controller MM reset input pin	SGDMA Controller MM and BD
mac_tx_rstn_o	CPU output reset pin	AXI Stream TX FIFO reset input pin	External Stream FIFO for CDC.
mac_rx_rstn_o	CPU output reset pin	AXI Stream RX FIFO reset input pin	External Stream FIFO for CDC.
lpddr4_rstn_o	CPU output reset pin	AXI IC M2 reset input pin	DDR Memory Controller

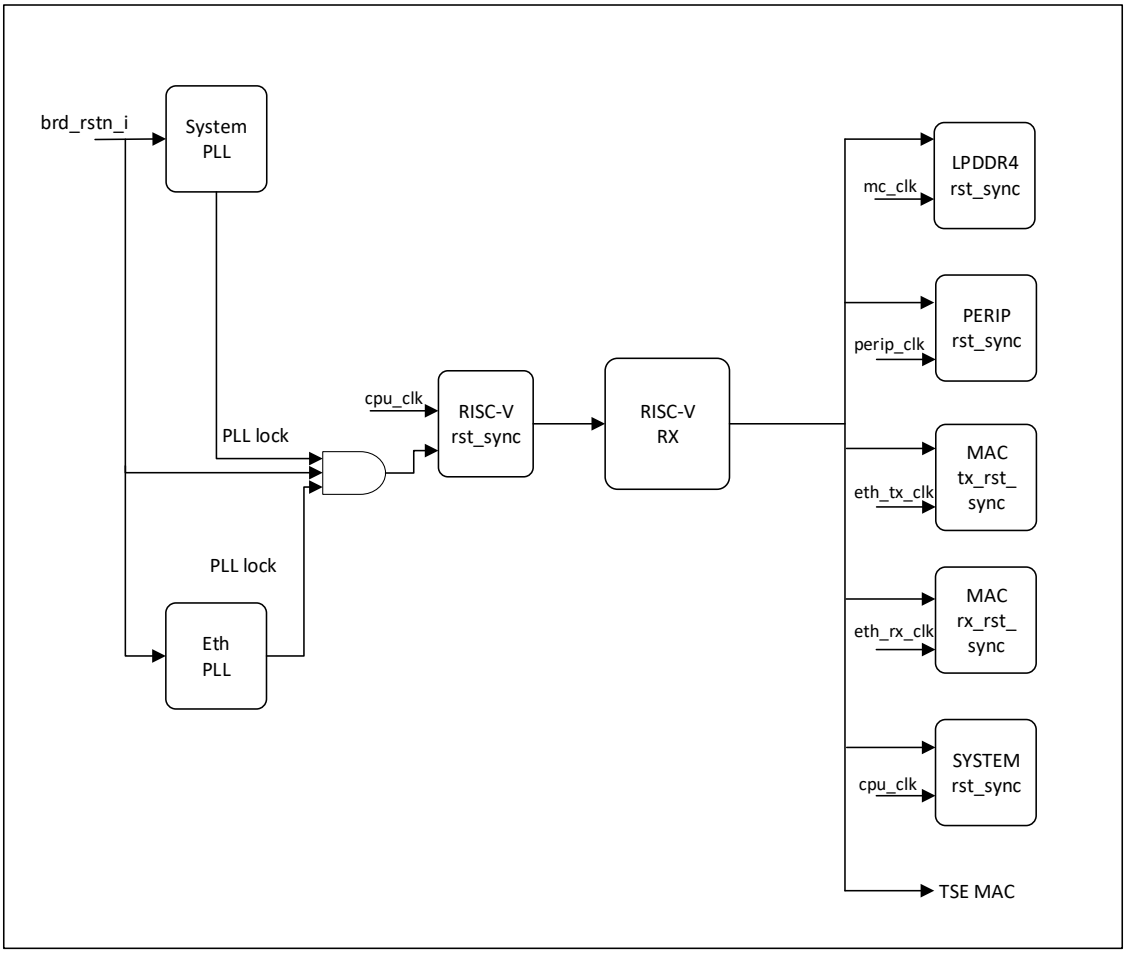


Figure 2.3. GHRD Reset Overview

2.4. GHRD Interrupts Overview

Table 2.4. GHRD Interrupt Overview

Sr No	RISC-V CPU IRQ Line	Interrupt Source
1	IRQ_2	SGDMA Controller MM2S IRQ
2	IRQ_3	SGDMA Controller S2MM IRQ
3	IRQ_4	TSEMAC IRQ
4	IRQ_5	OSPI IRQ
5	IRQ_6	GPIO IRQ

Sr No	RISC-V CPU IRQ Line	Interrupt Source
6	IRQ_7	LPDDR IRQ
7	IRQ_8	UART IRQ

For more information about the platform level interrupt controller information, refer to the Platform Level Interrupt Controller section in the RISC-V RX CPU IP User Guide (FPGA-IPUG-02280) document.

2.5. IP Configurations

The reference design is created using Lattice Propel Builder. The top-level HDL file is generated by Propel Builder and is used as the top module for the design. The design parameterization is performed by configuring the IP in Propel Builder. This section describes the following IPs and their configuration.

2.5.1. RISC-V RX CPU

The RISC-V RX CPU IP has AXI-based instruction and data ports. The instruction ports are connected to the memory that contains the bootloader software or the FreeRTOS application software for CPU execution. The data port is connected to memory and peripherals for control.

For more information about the IP core including memory map information, refer to RISC-V RX CPU IP User Guide (FPGA-IPUG-02280).

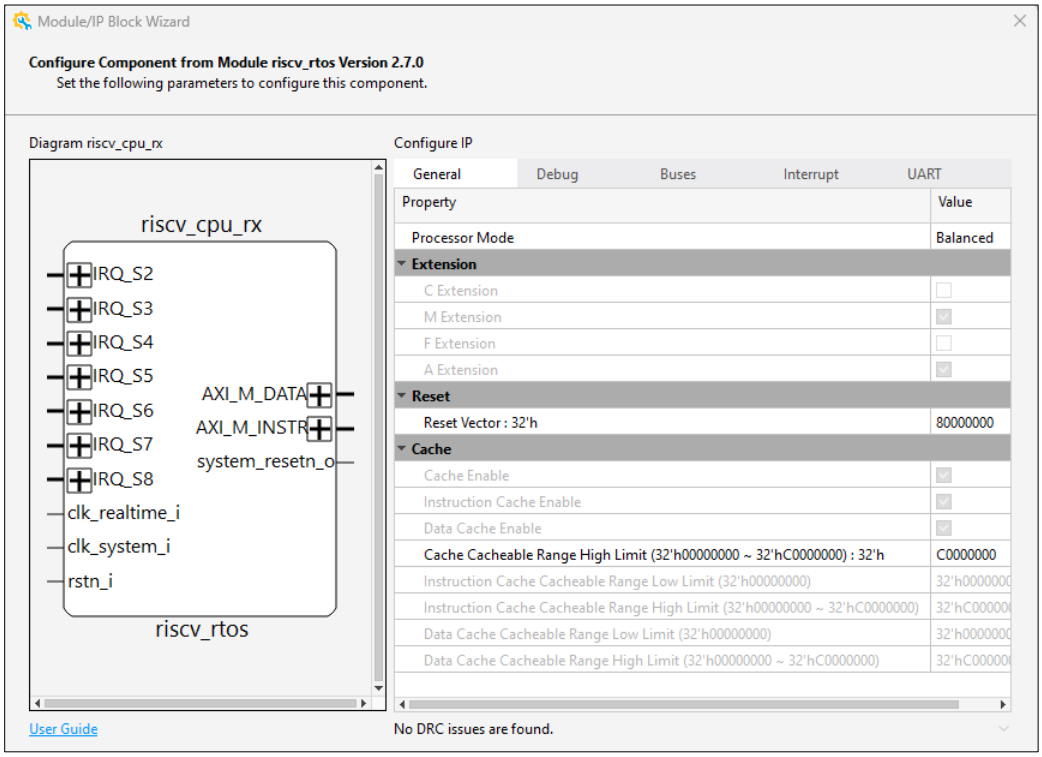


Figure 2.4. RISC-V RX CPU IP Configuration – General

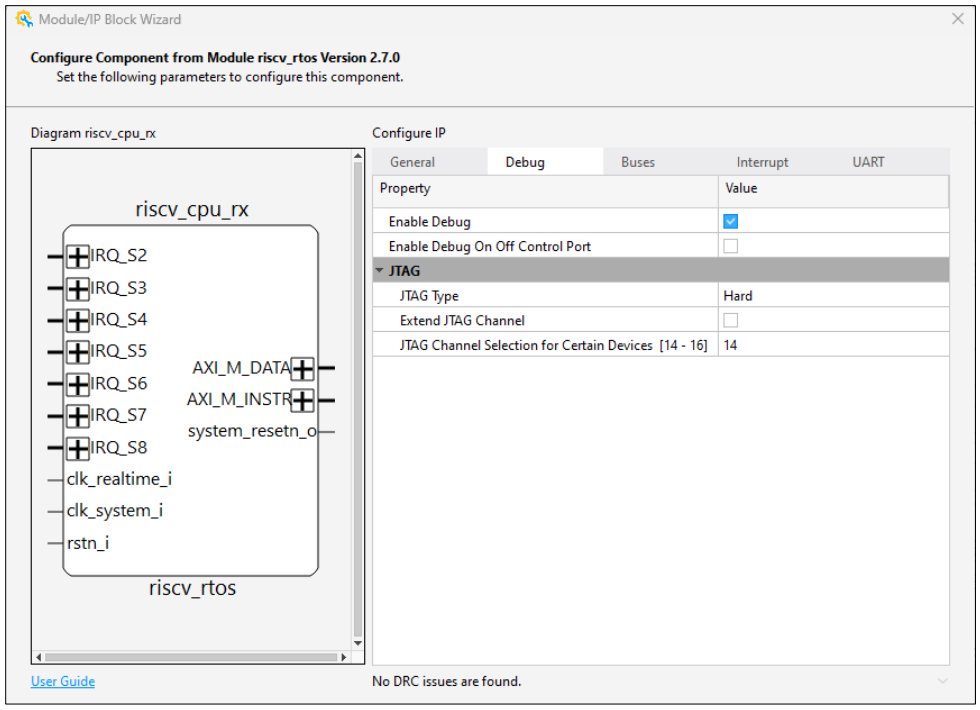


Figure 2.5. RISC-V RX CPU IP Configuration – Debug

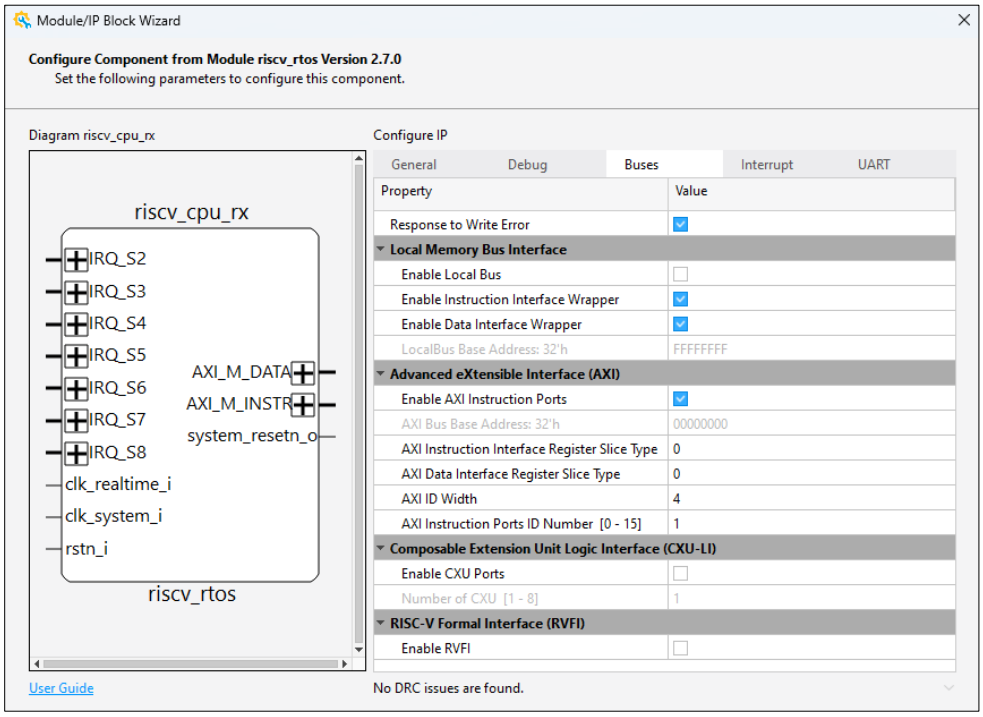


Figure 2.6. RISC-V RX CPU IP Configuration – Buses

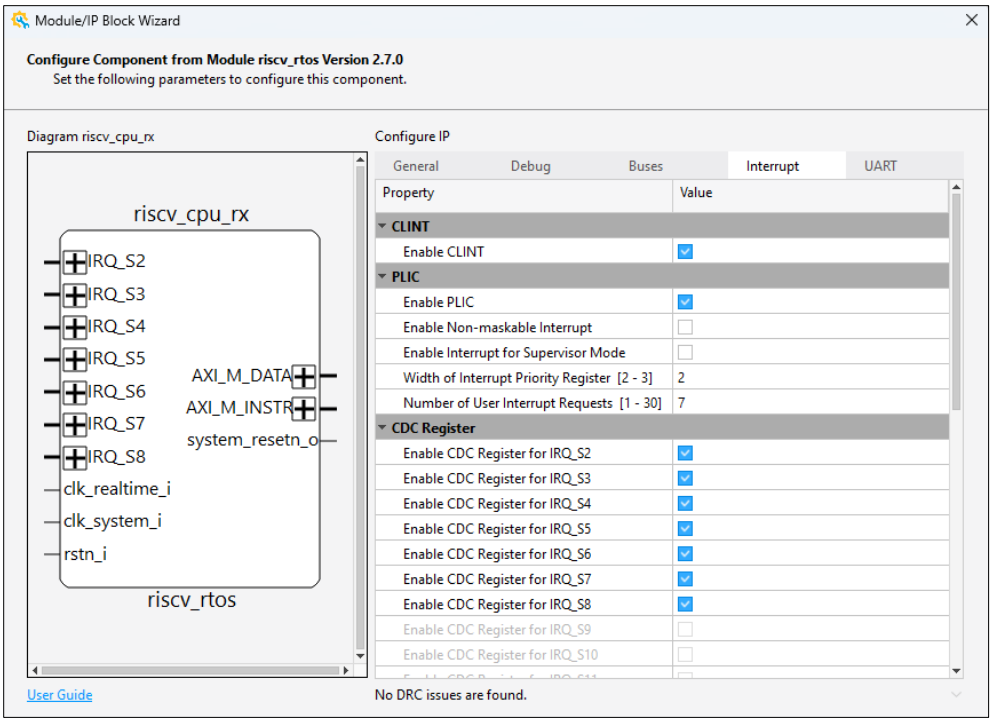


Figure 2.7. RISC-V RX CPU IP Configuration – Interrupt

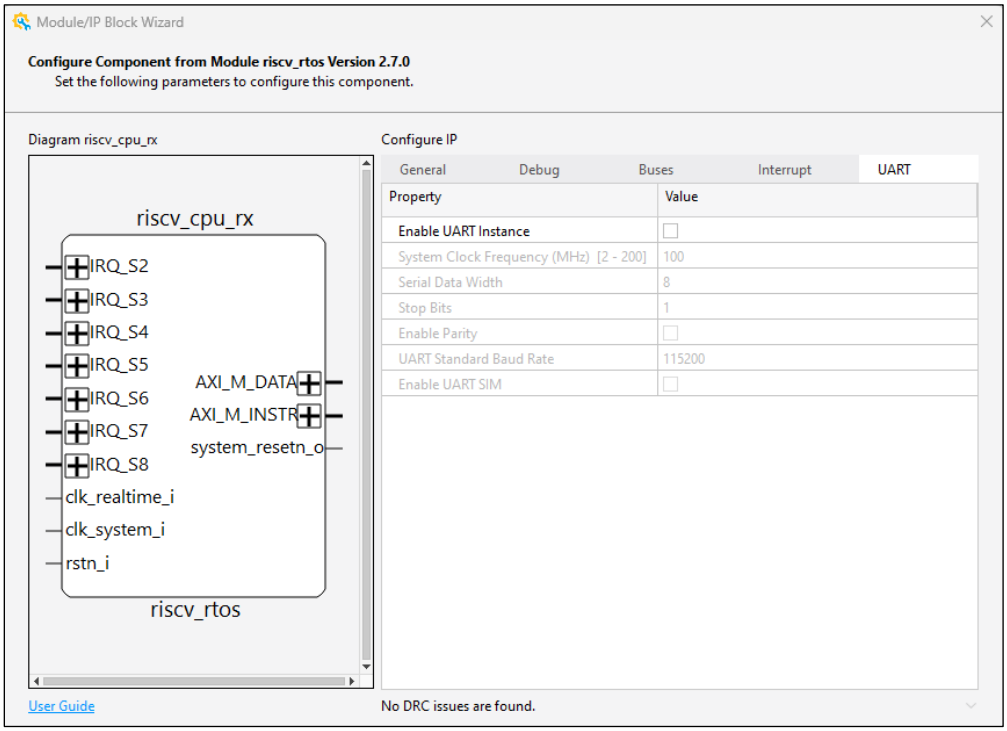


Figure 2.8. RISC-V RX CPU IP Configuration – UART

2.5.2. AXI Interconnect

The AXI Interconnect IP is a key component in system that uses the AMBA AXI4 protocol. It acts as a communication hub between multiple AXI managers and subordinates, enabling efficient data transfer and system scalability.

For more information about the IP core, refer to AXI Interconnect IP User Guide (FPGA-IPUG-02196).

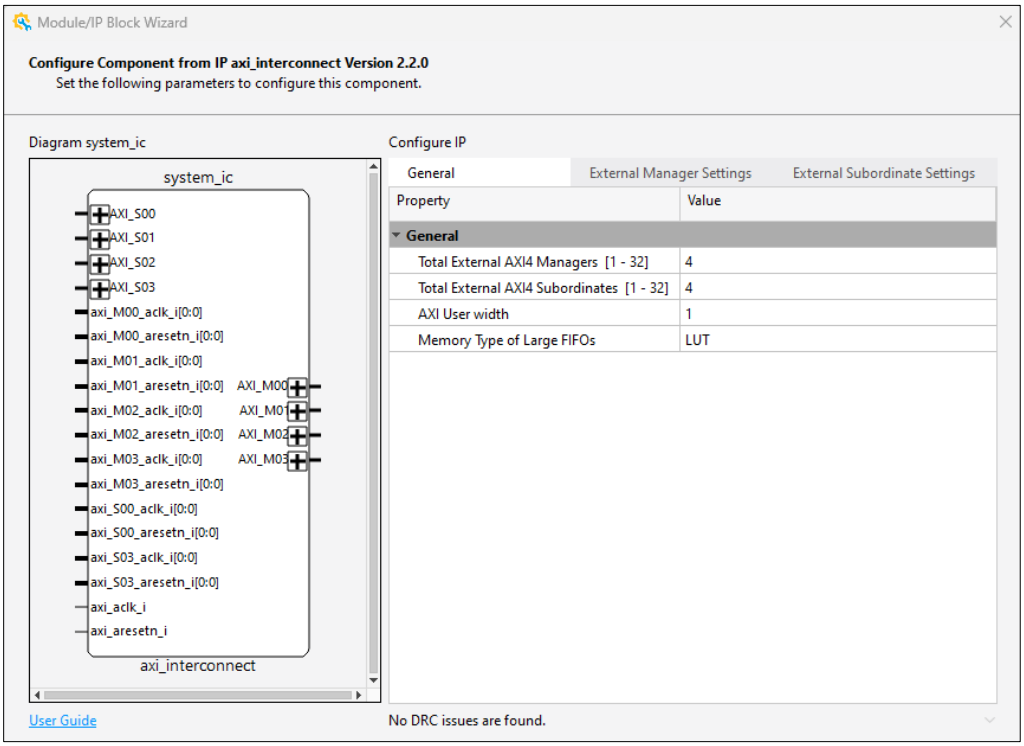


Figure 2.9. AXI Interconnect IP Configuration – General

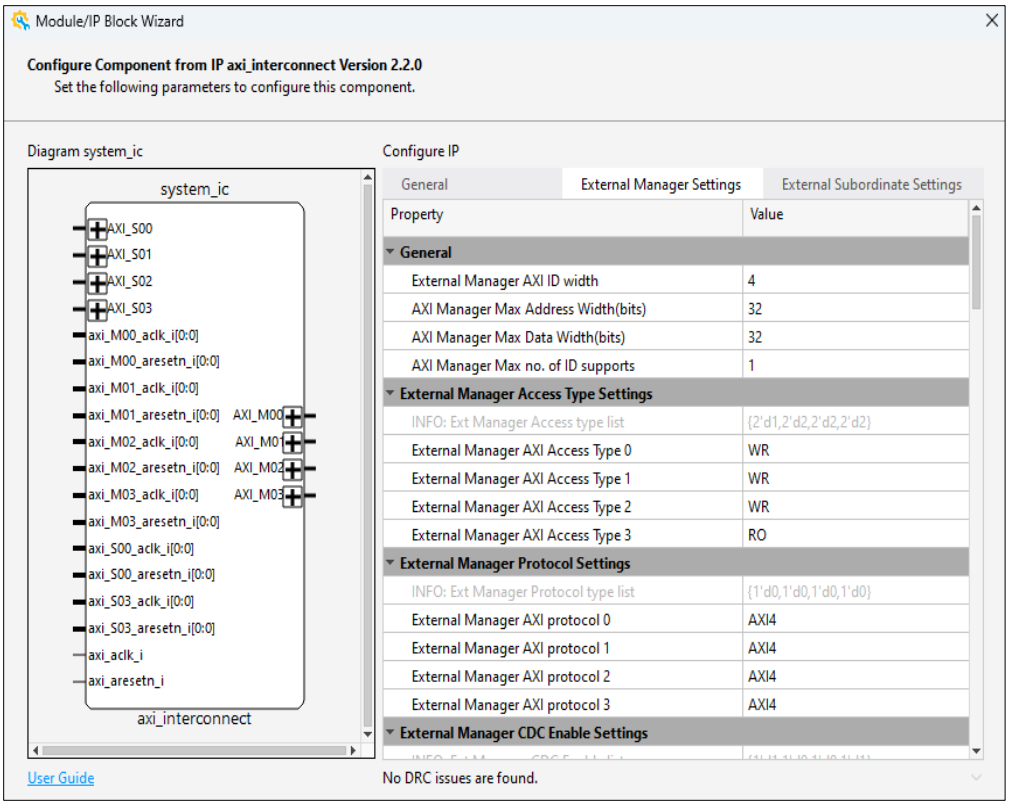


Figure 2.10. AXI Interconnect IP Configuration – External Manager Settings

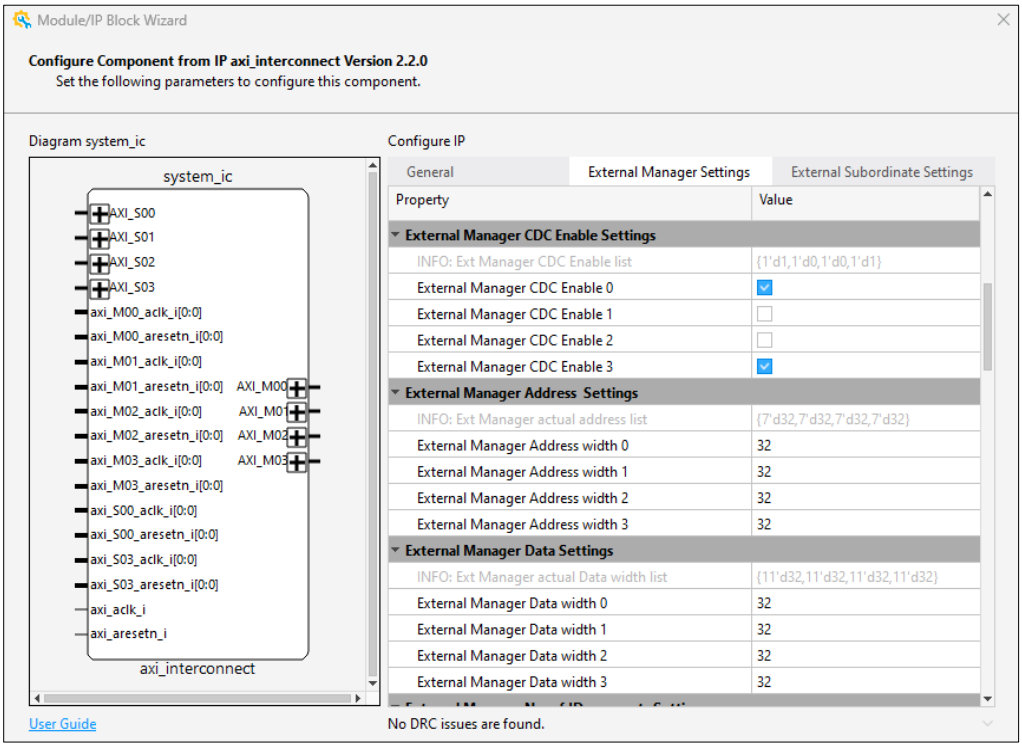


Figure 2.11. AXI Interconnect IP Configuration – External Manager Settings

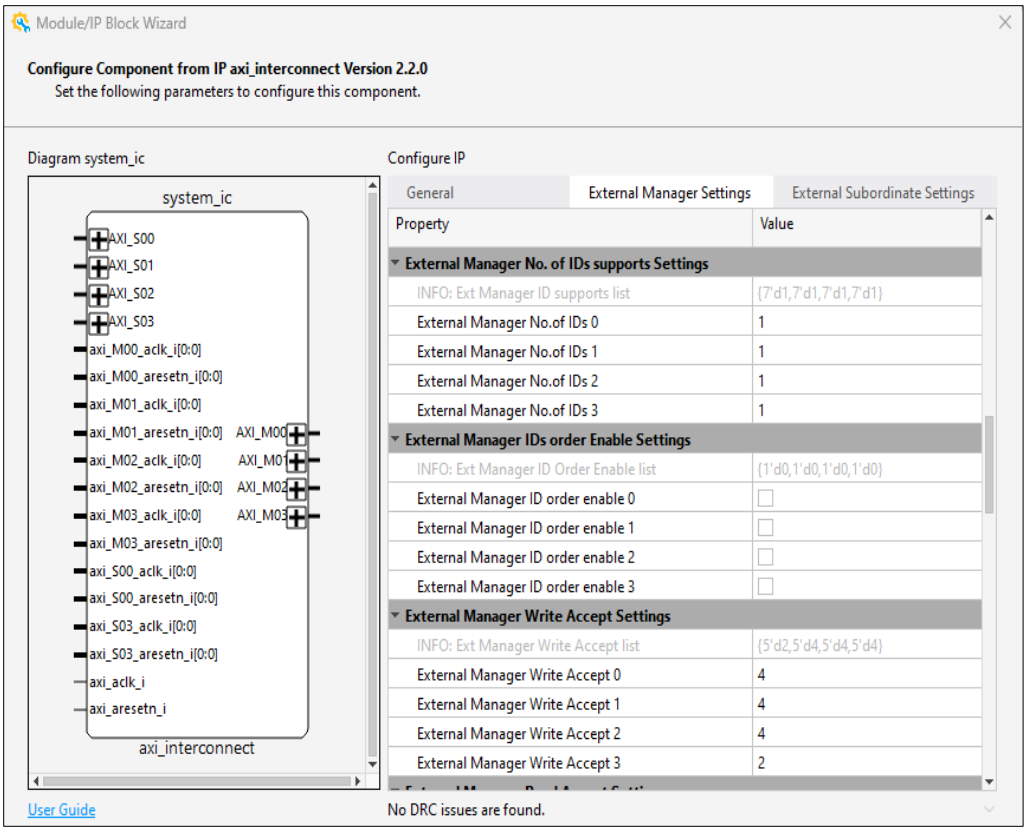


Figure 2.12. AXI Interconnect IP Configuration – External Manager Settings

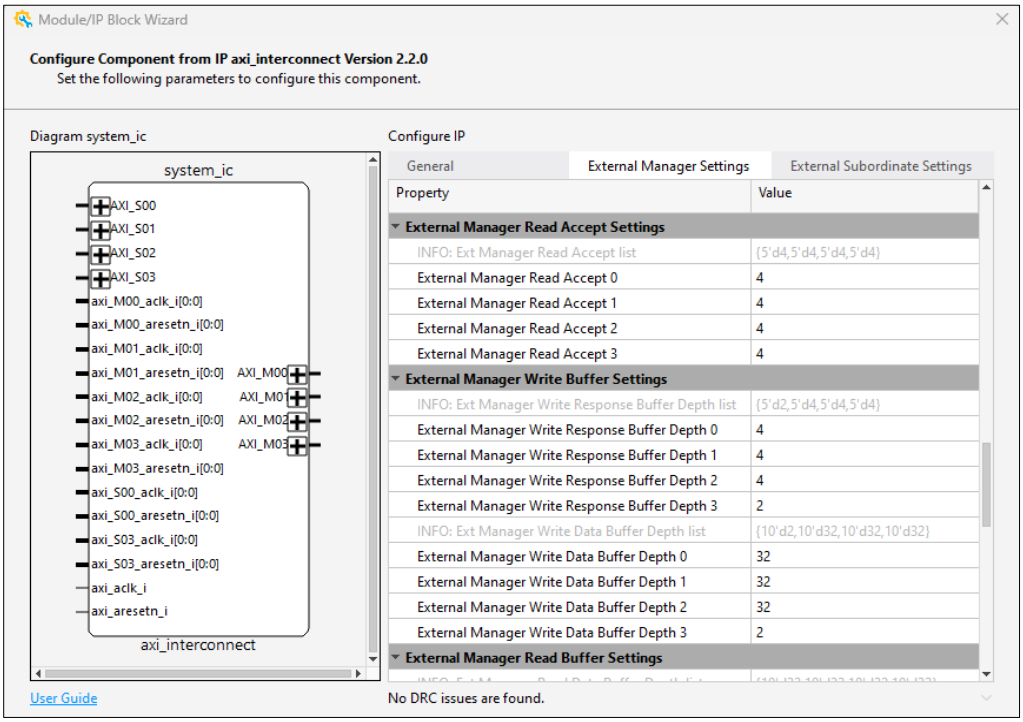


Figure 2.13. AXI Interconnect IP Configuration – External Manager Settings

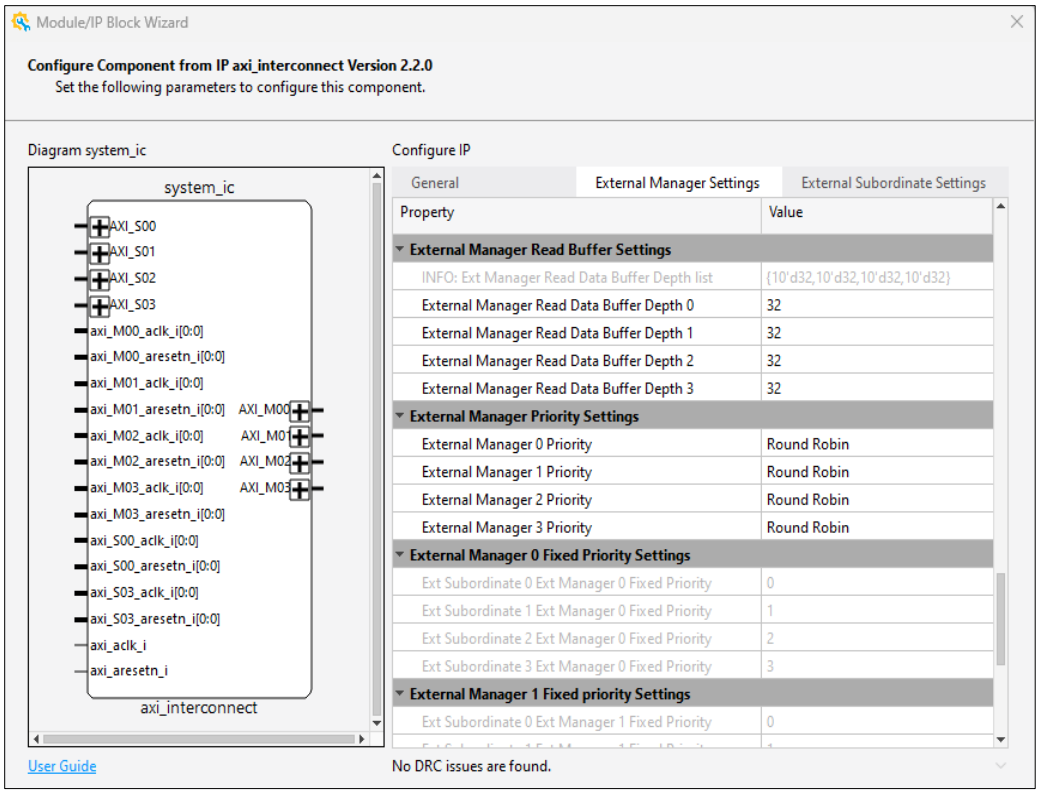


Figure 2.14. AXI Interconnect IP Configuration – External Manager Settings

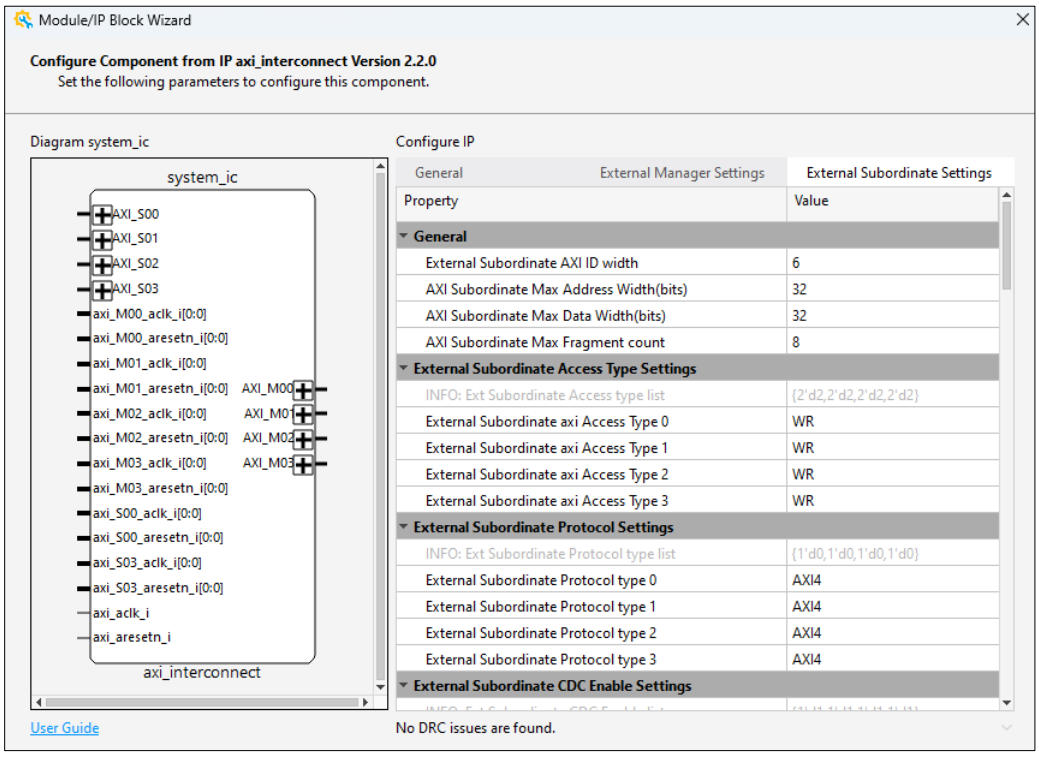


Figure 2.15. AXI Interconnect IP Configuration – External Subordinate Settings

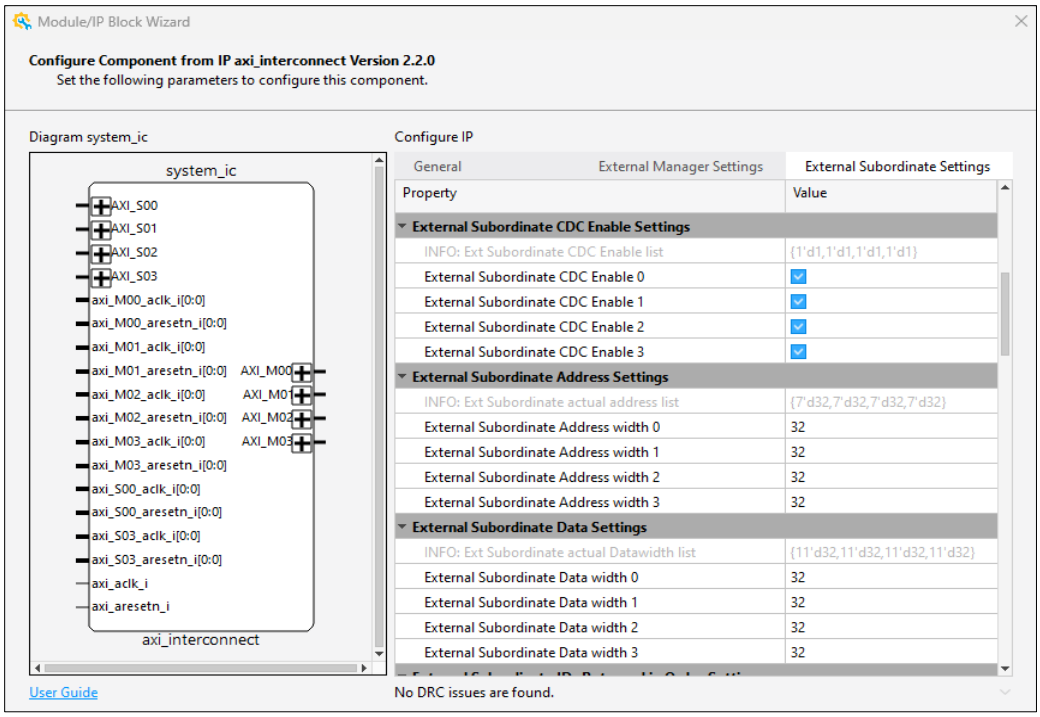


Figure 2.16. AXI Interconnect IP Configuration – External Subordinate Settings

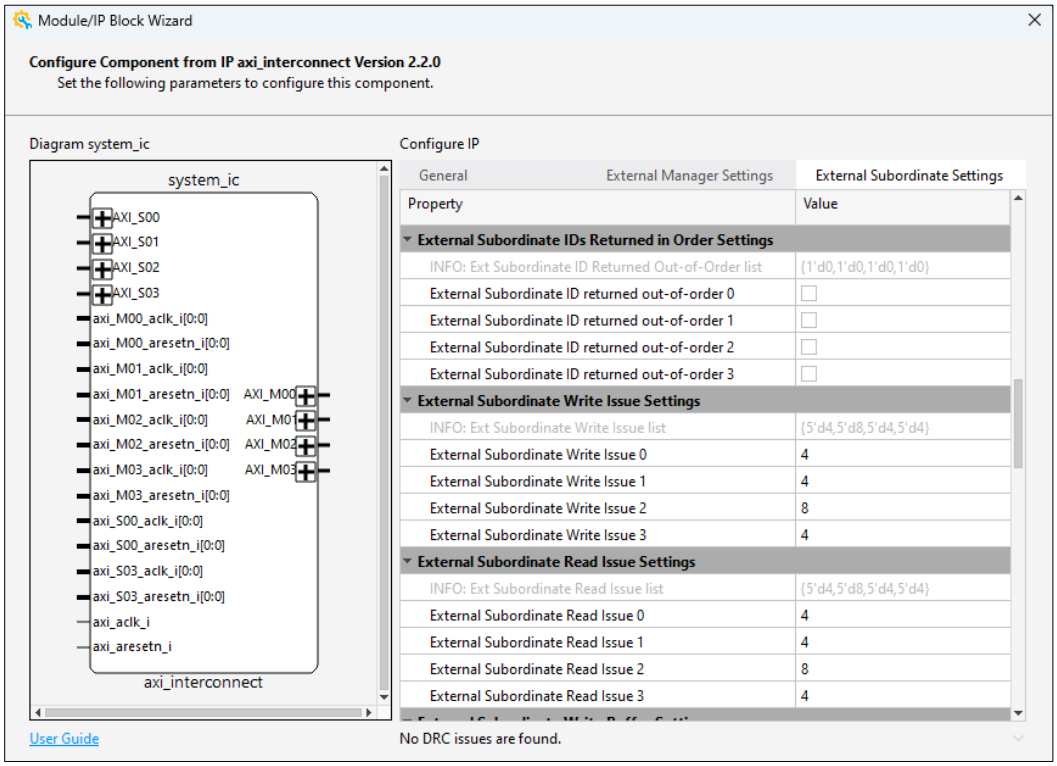


Figure 2.17. AXI Interconnect IP Configuration – External Subordinate Settings

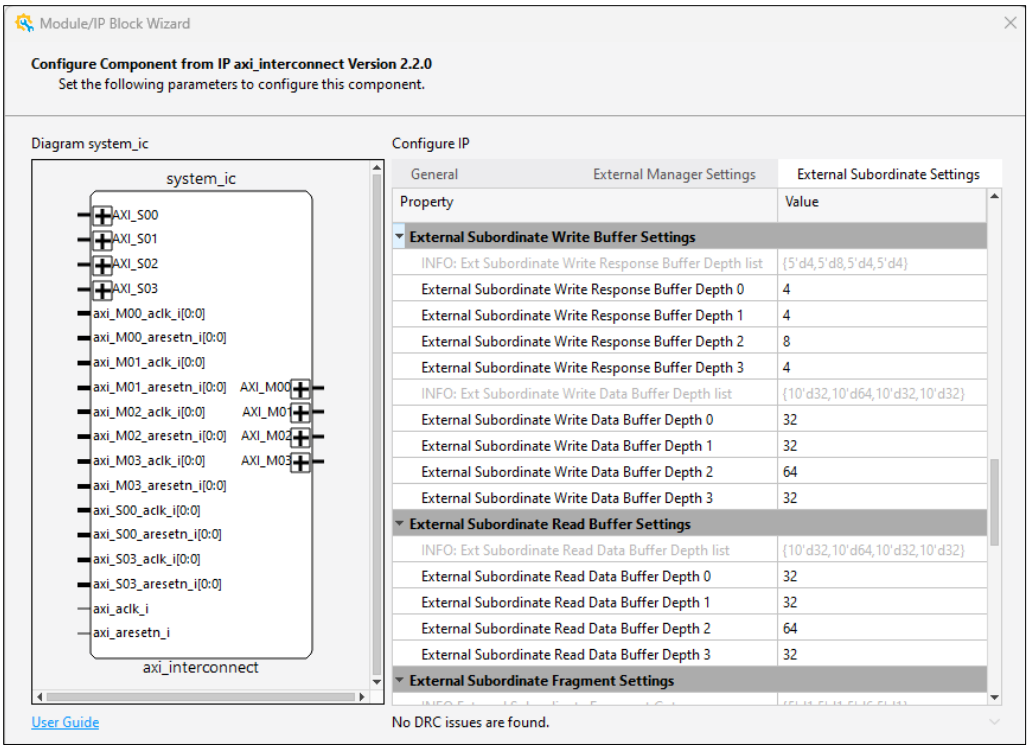


Figure 2.18. AXI Interconnect IP Configuration – External Subordinate Settings

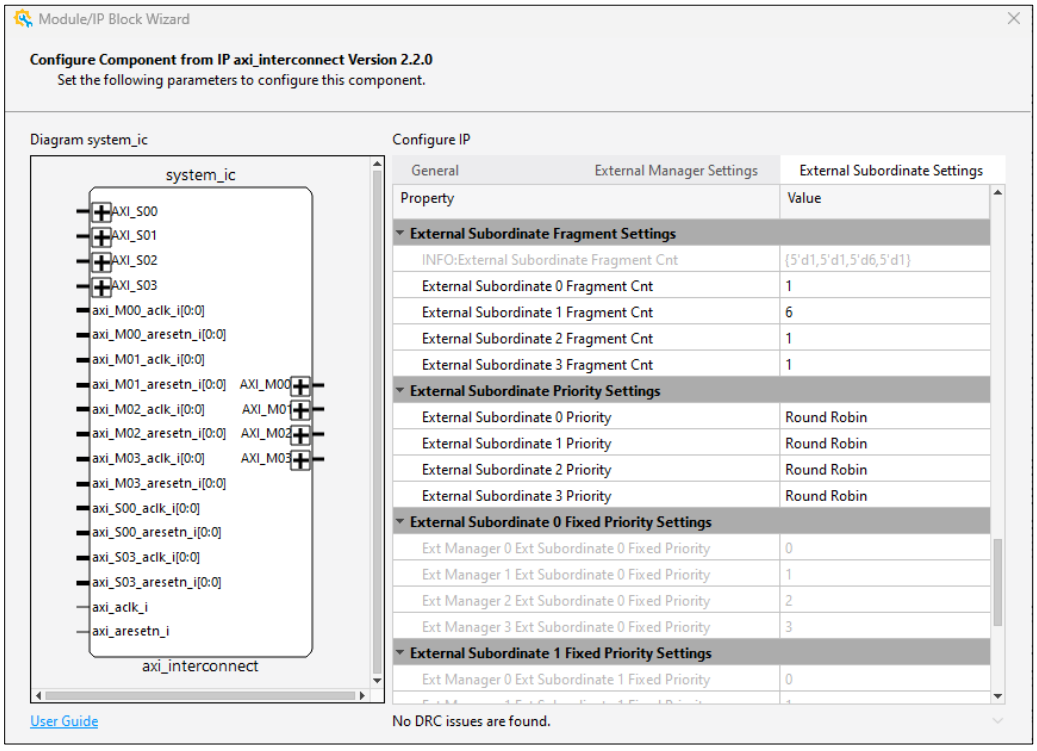


Figure 2.19. AXI Interconnect IP Configuration – External Subordinate Settings

2.5.3. APB Interconnect

The APB Interconnect IP is used in system for connecting low-bandwidth, low-power peripherals to the main system bus.

For more information about the IP core, refer to APB Interconnect IP User Guide (FPGA-IPUG-02054).

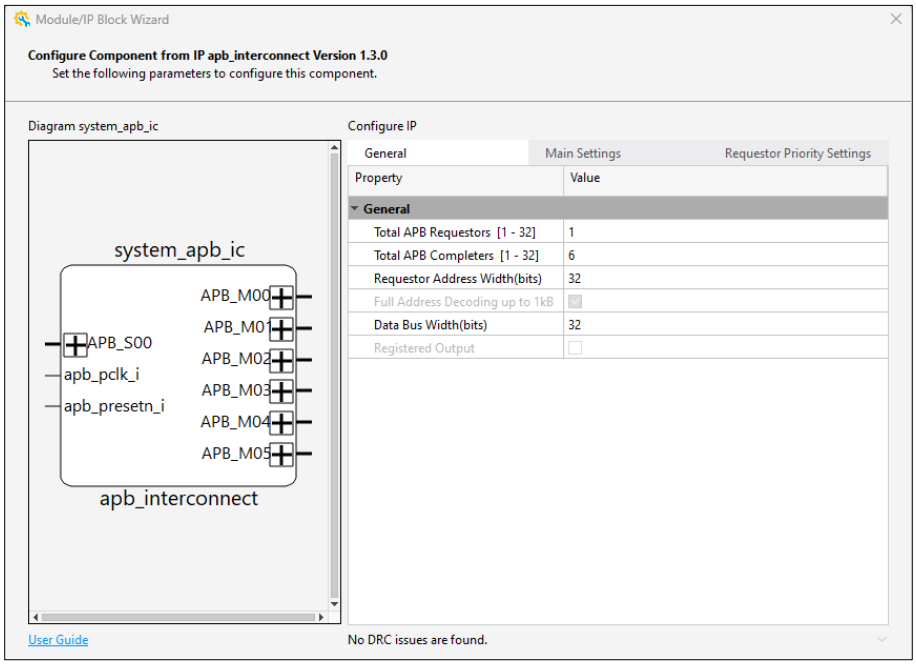


Figure 2.20. APB Interconnect IP Configuration – General

2.5.4. AXI4 to APB Bridge

The AXI4 to APB Bridge IP is used to convert the AXI4 bus transaction into low-speed APB bus transaction. For more information about the IP, refer to AXI4APB Bridge IP User Guide (FPGA-IPUG-02198).

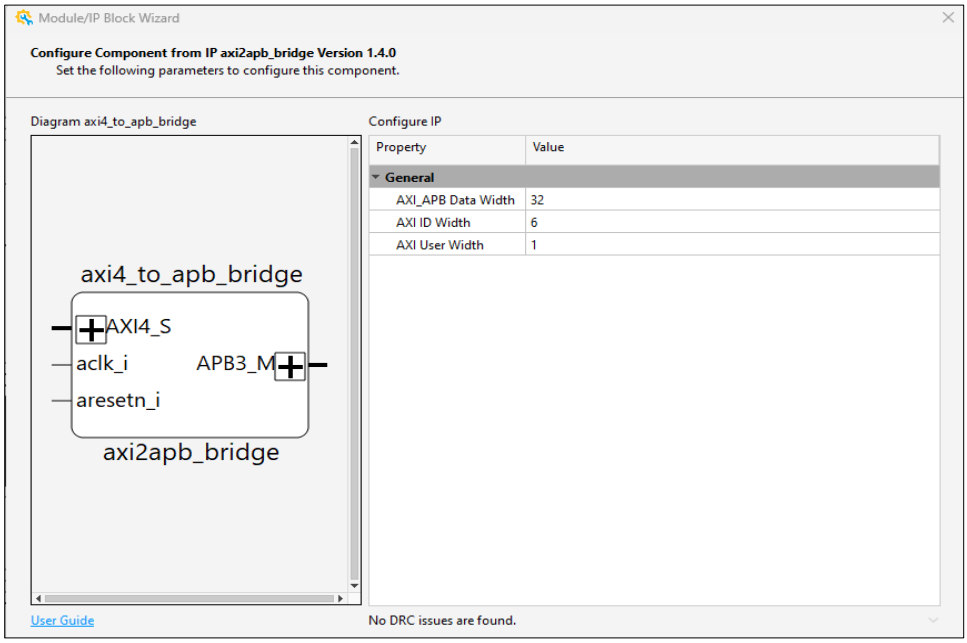


Figure 2.21. AXI4 to APB Bridge IP Configuration – General

2.5.5. DDR Memory Controller

The DDR Memory Controller IP enables access to the external LPDDR4 memory devices. The memory can be used to store CPU software code and data as well as Ethernet packet data.

For more information about the IP core including register map information, refer to DDR Memory Controller IP Core (FPGA-IPUG-02208).

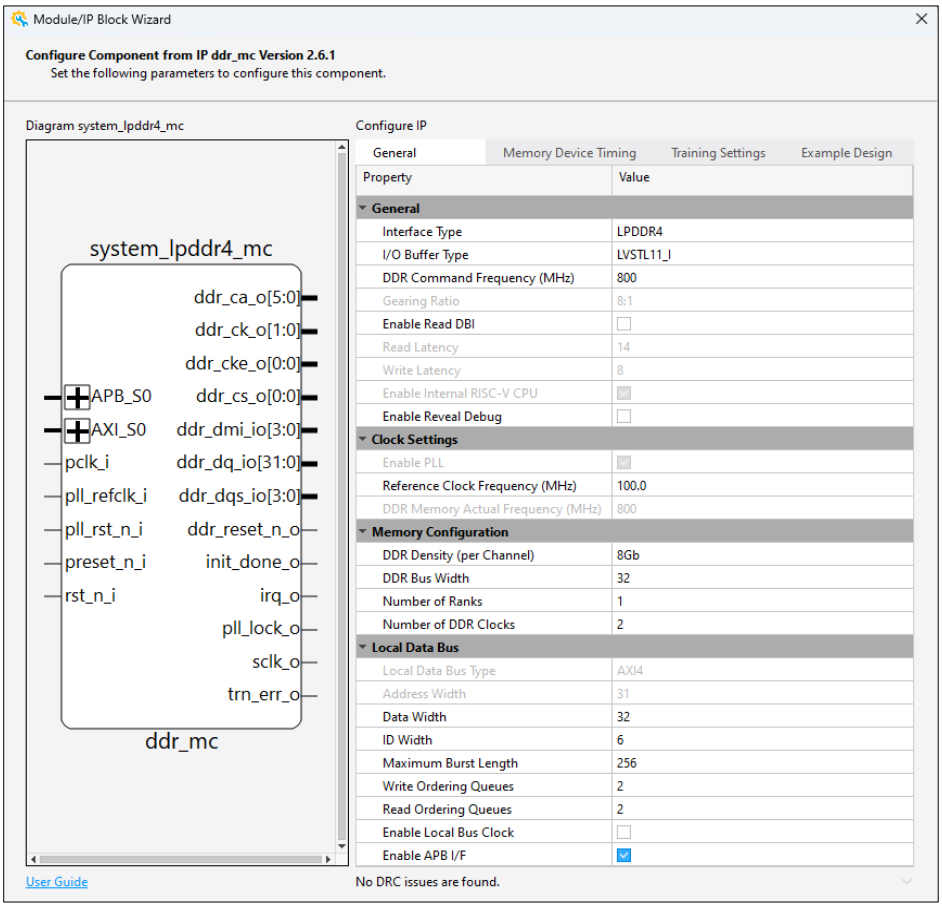


Figure 2.22. DDR Memory Controller IP Configuration – General

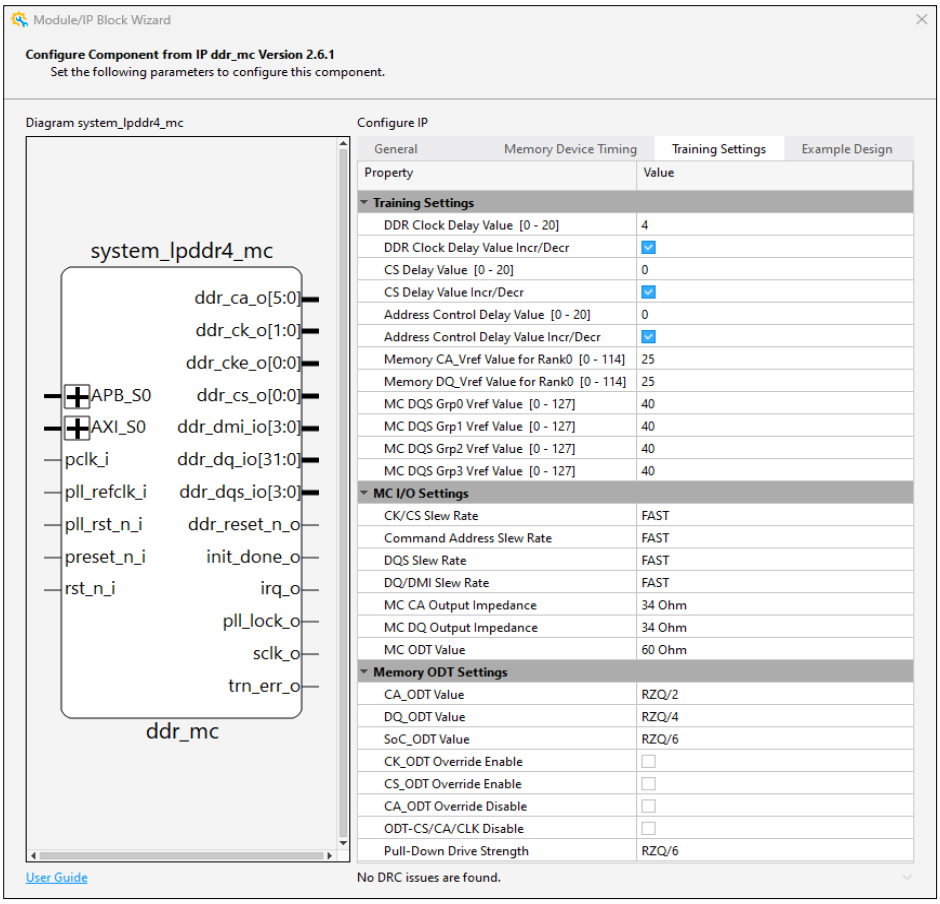


Figure 2.23. DDR Memory Controller IP Configuration – Training Settings

2.5.6. Octal SPI Controller

The Octal SPI is an eight tri-state data line serial interface that is commonly used to store, program, erase, and read SPI flash memories. Octal SPI enhances the throughput of a standard SPI by eight times since eight bits are transferred every cycle. In GSRD, Quad SPI flash is used to store application software and bitstreams for both Primary and Golden systems. Hence, the Octal SPI Controller IP is configured in a four tri state data line serial interface.

For more information about the IP core including register map information, refer to Octal SPI Controller IP User Guide (FPGA-IPUG-02273).

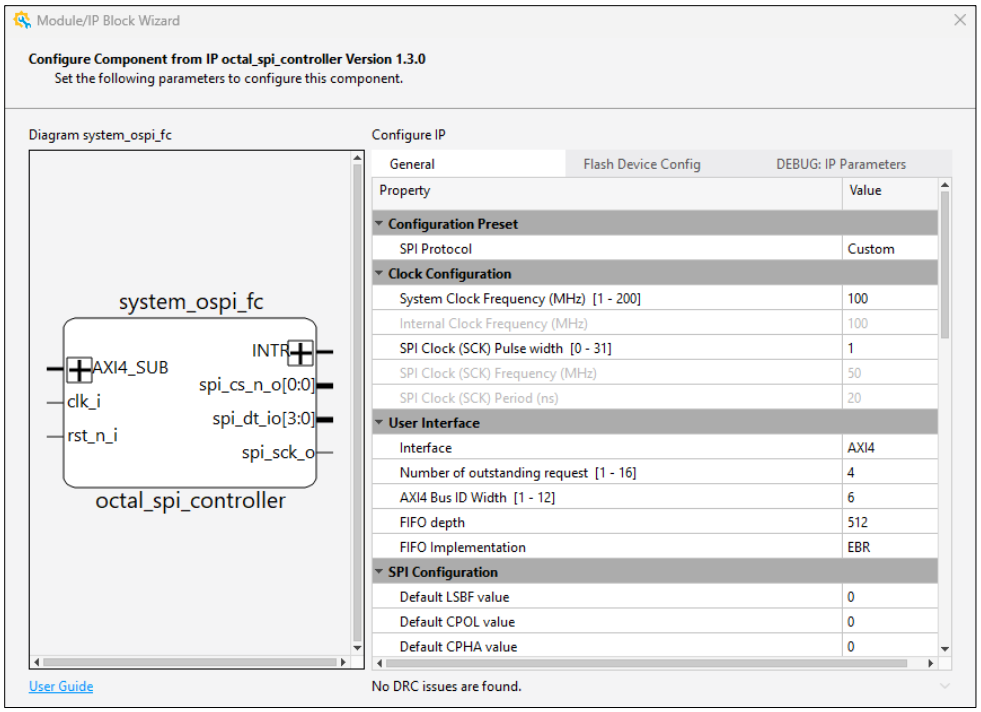


Figure 2.24. Octal SPI Controller IP Configuration – General

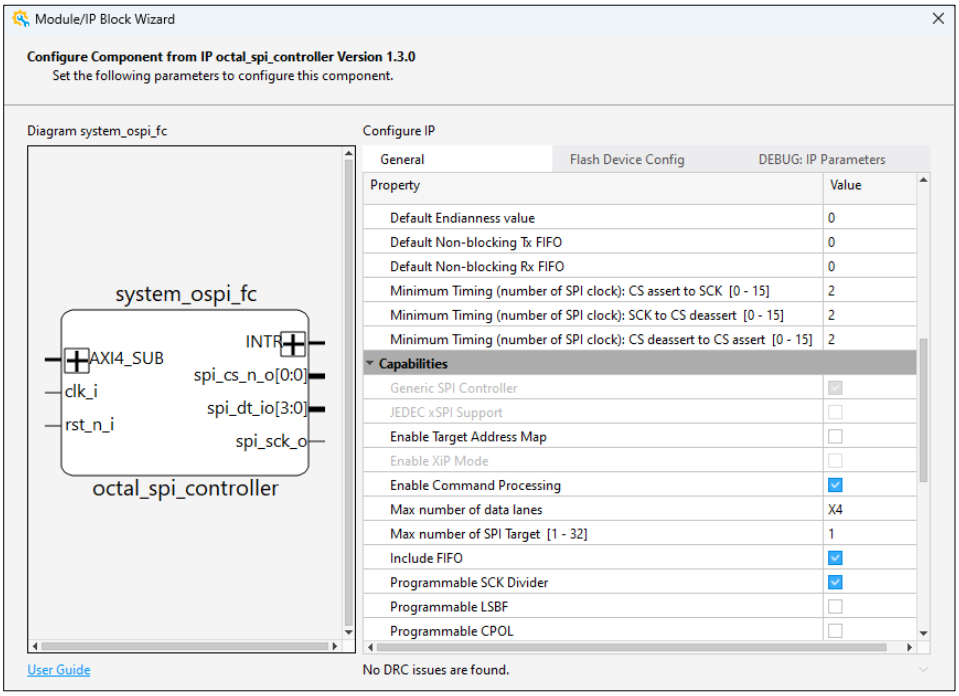


Figure 2.25. Octal SPI Controller IP Configuration – General

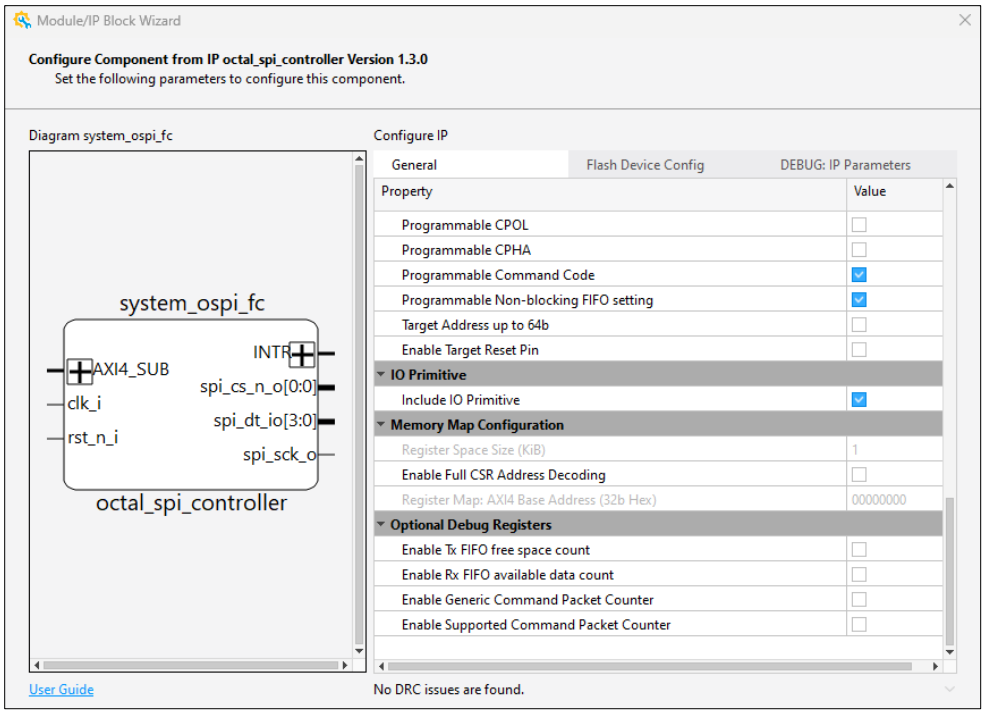


Figure 2.26. Octal SPI Controller IP Configuration – General

2.5.7. Tri-Speed Ethernet MAC + RGMII

The Tri-Speed Ethernet (TSE) IP solution consists of the TSE IP Media Access Controller (MAC) core and the RGMII interface. The TSE IP is a complex core containing all the necessary logic, interfacing, and clocking infrastructure to allow integrating an external industry-standard Ethernet PHY with an internal processor, with minimal overhead. The Avant-E GSRD supports 1 Gbps network interface as per the IEEE 802.3 standard.

For more information about the IP core including register map information, refer to Tri-Speed Ethernet MAC IP User Guide (FPGA-IPUG-02084).

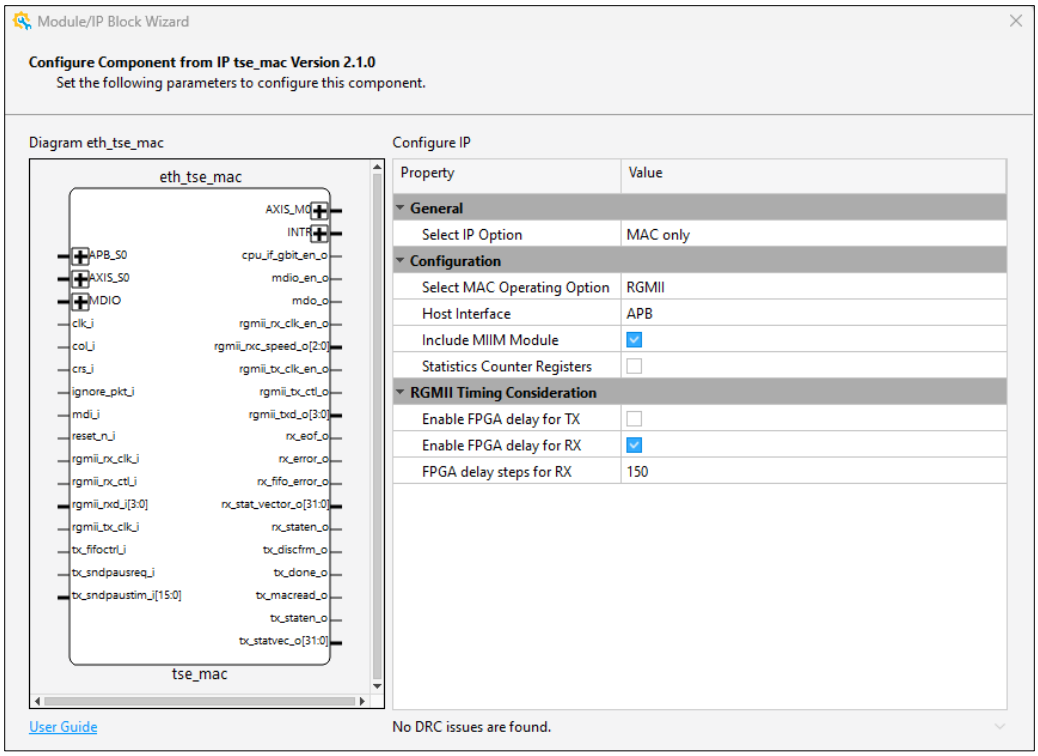


Figure 2.27. TSE IP Configuration

2.5.8. Scatter-Gather DMA Controller

The SGDMA Controller IP core is used to access the main memory independent of the CPU processor. It offloads processor intervention. The processor initiates transfer to SGDMA Controller and receive interrupt on completion of the transfer by the DMA engine. The core implements a configurable, AXI4-compliant DMA controller with scatter-gather capability. It also implements the AXI4-Stream interface to support stream data from TSE MAC. The APB CSR interface is used to configure the control and status registers by the RISC-V CPU.

For more information about the IP core including register map information, refer to SGDMA Controller IP User Guide (FPGA-IPUG-02131).

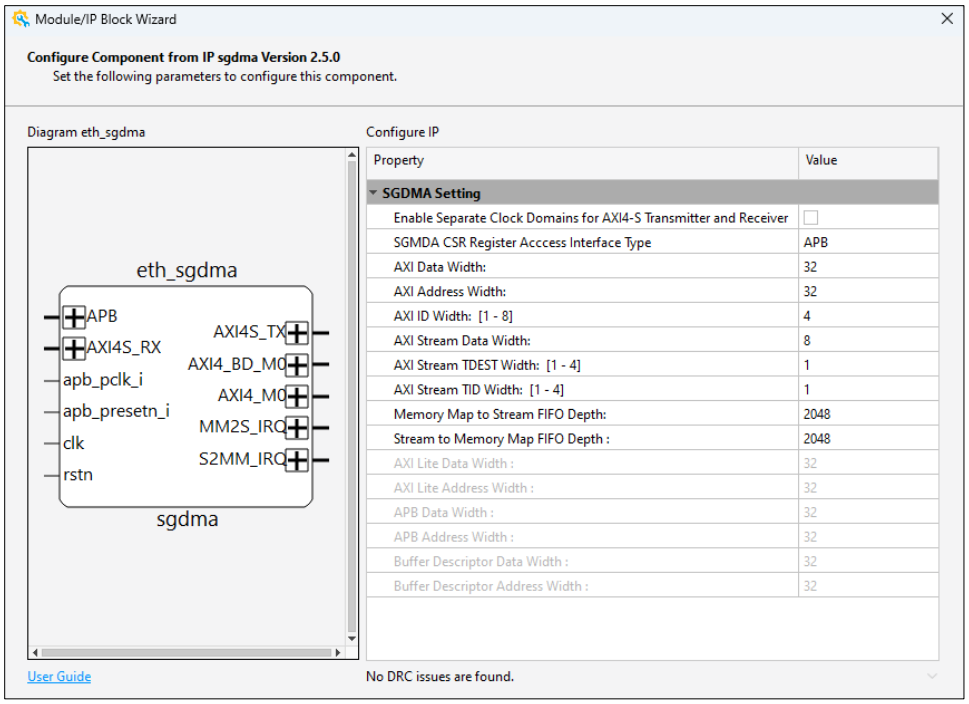


Figure 2.28. SGDMA Controller IP Configuration

2.5.9. UART

The Universal Asynchronous Receiver/Transmitted (UART) Transceiver IP core performs serial-to-parallel conversion of data characters received from a peripheral UART device and parallel-to-serial conversion of data characters received from the host locater insider the FPGA through an APB interface.

For more information about the IP core including register map information, refer to UART IP User Guide (FPGA-IPUG-02105)

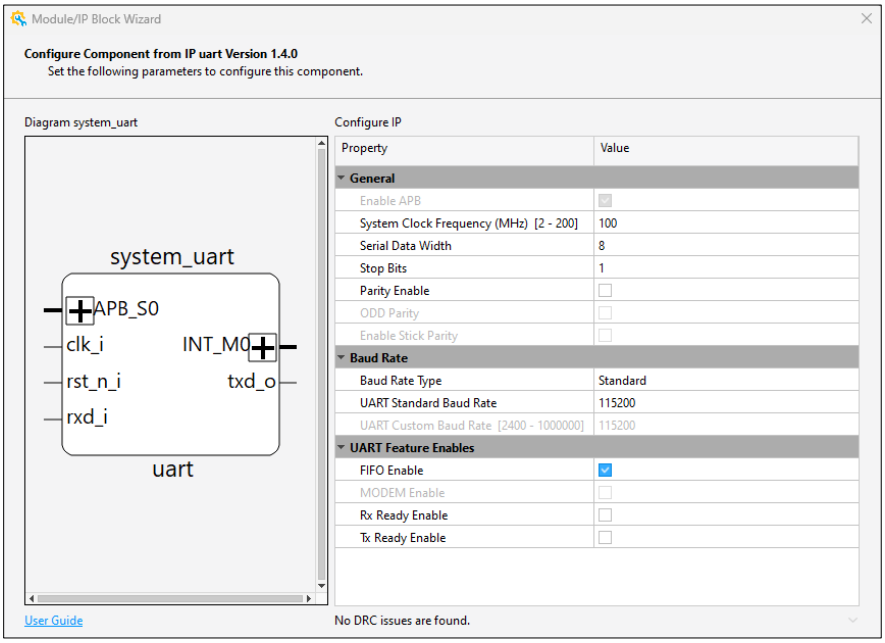


Figure 2.29. UART IP Configuration

2.5.10. GPIO

The General-Purpose Input/Output (GPIO) peripheral IP provides dedicated memory-mapped interface to configure the GPIO ports as well as the number of input and output ports.

For more information about the IP core including register map information, refer to GPIO IP User Guide (FPGA-IPUG-02076).

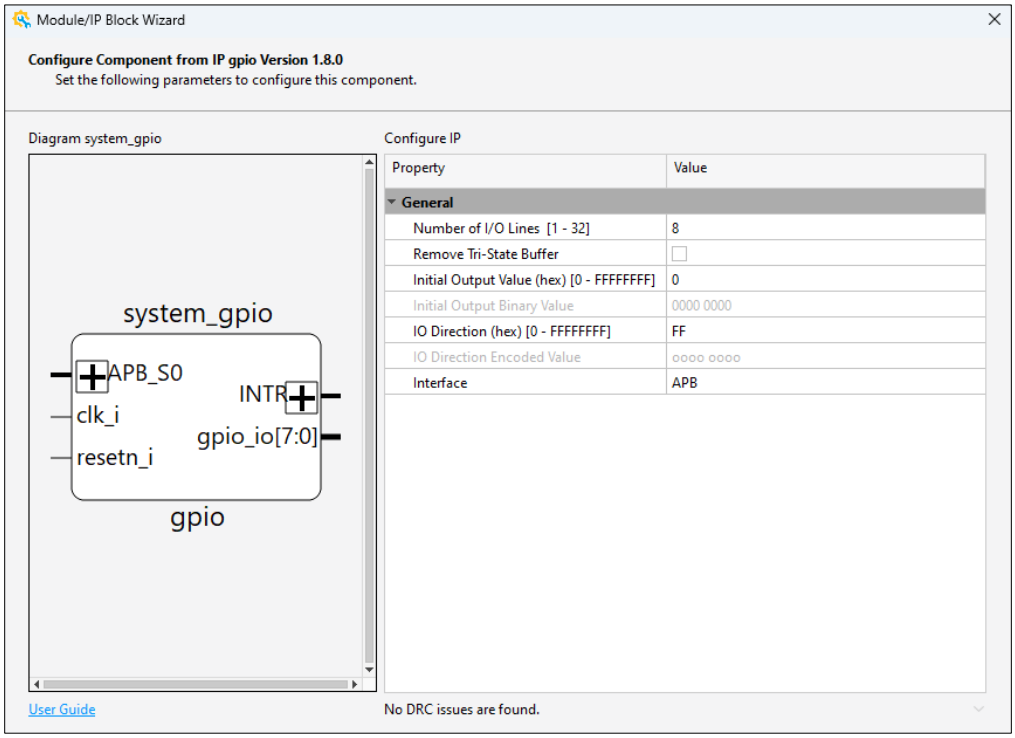


Figure 2.30. GPIO IP Configuration

2.5.11. Multi-Boot Configuration Module

The Multi-Boot Configuration is used to trigger an internal FPGA REFRESH/PROGRAMN command to LMMI logic. This IP implements an APB endpoint which decodes the RISC-V CPU command data. The LMMI host FSM used inside IP to execute the soft reset to load the next or alternate bitstream and application software data onto the FPGA.

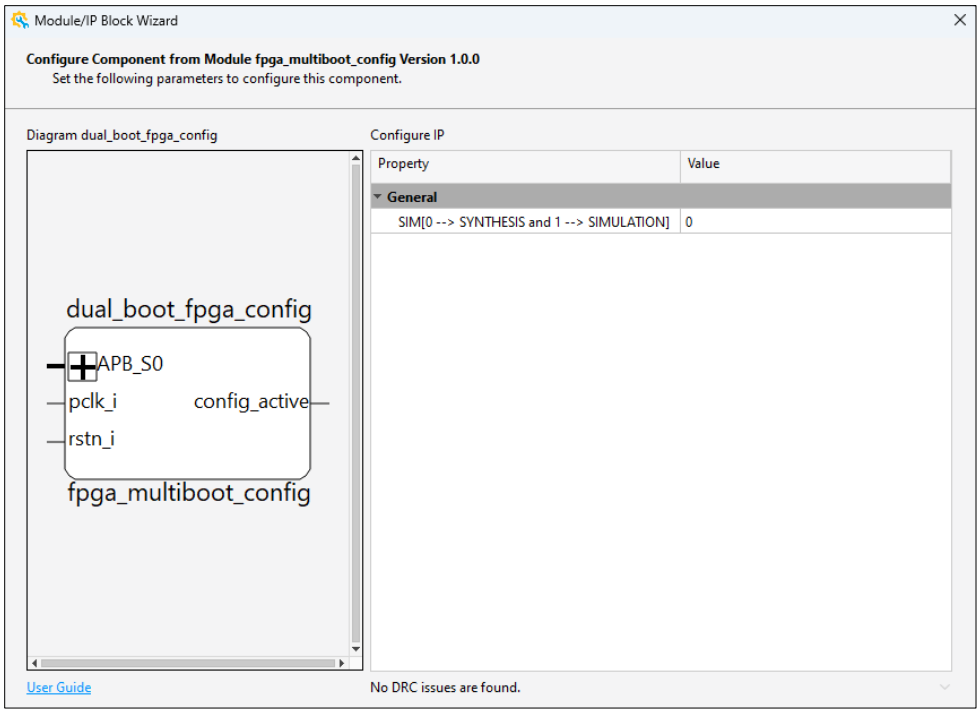


Figure 2.31. Multi-Boot Module Configuration

2.5.12. System Memory

The System Memory implements EBR or Distributed Memory in either single port or dual port AHB-L or AXI4 subordinate. In GSRD, System Memory is configured with EBR as the memory type and single port AXI4 as the interface. The System Memory in this design is used to store the bootloader. The System Memory size is 256 kB.

For more information about the IP, refer to System Memory IP User Guide (FPGA-IPUG-02073).

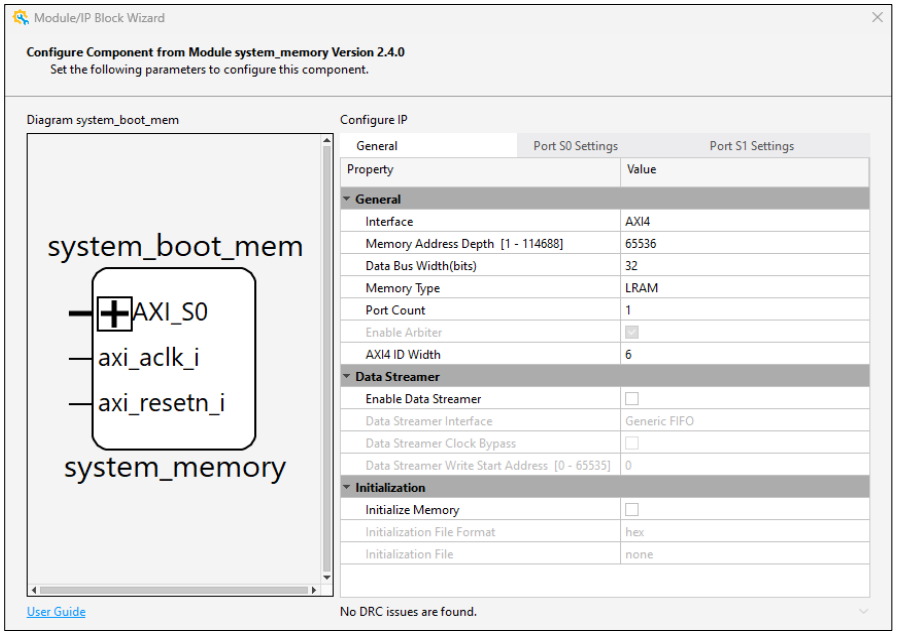


Figure 2.32. System Memory IP Configuration – General

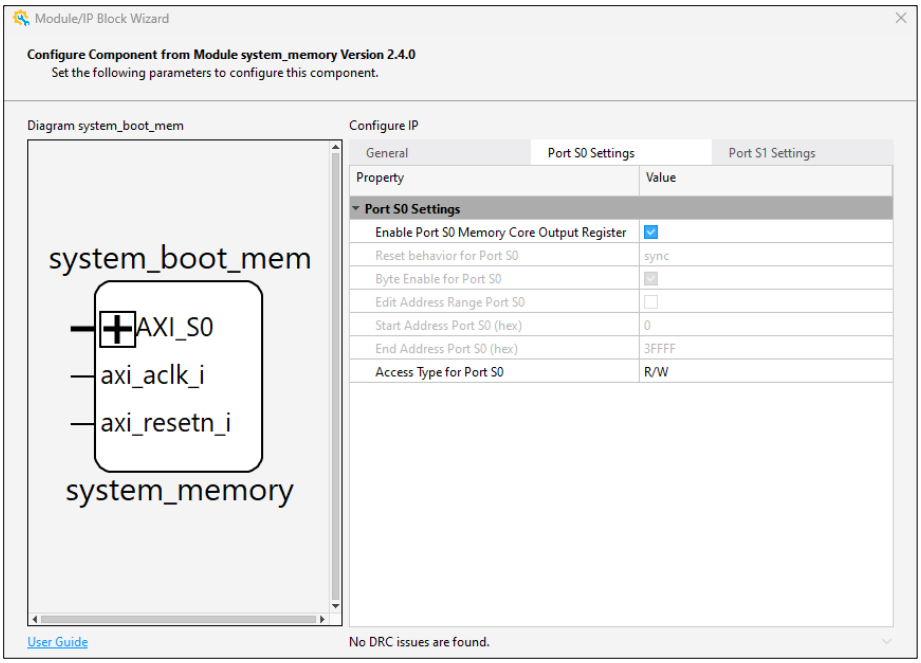


Figure 2.33. System Memory IP Configuration – Port S0 Settings

2.5.13. PLL

The PLL IP is used to generate multiple clocks used in the system. There are two PLL instances used in GSRD, system_pll and eth_tx_pll.

For more information about the IP, refer to Avant PLL Module User Guide (FPGA-IPUG-02220).

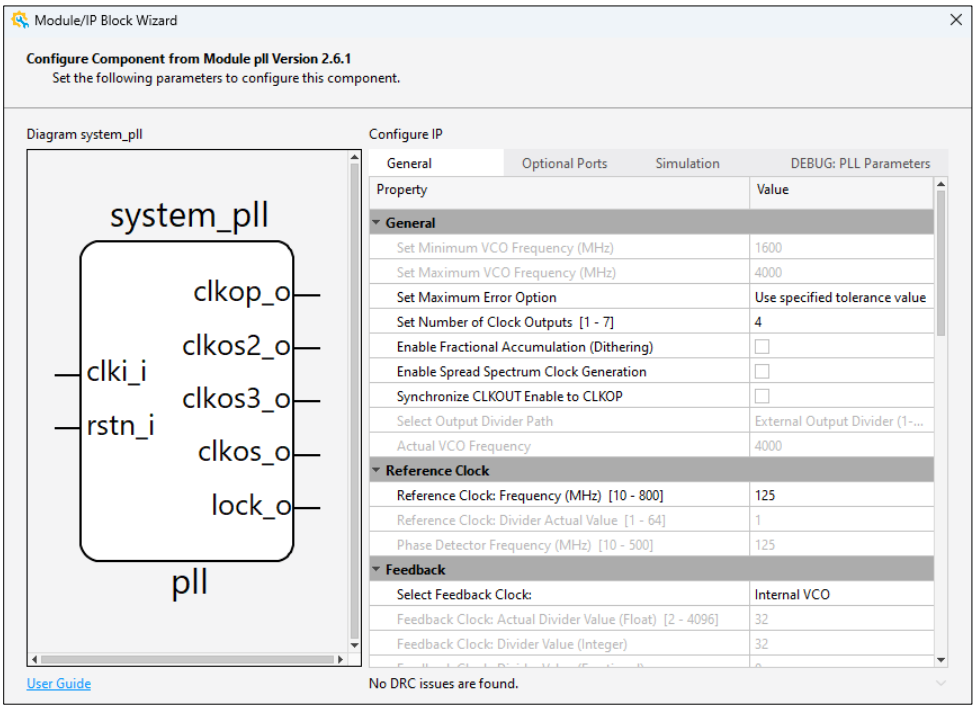


Figure 2.34. PLL IP Configuration -General for system_pll

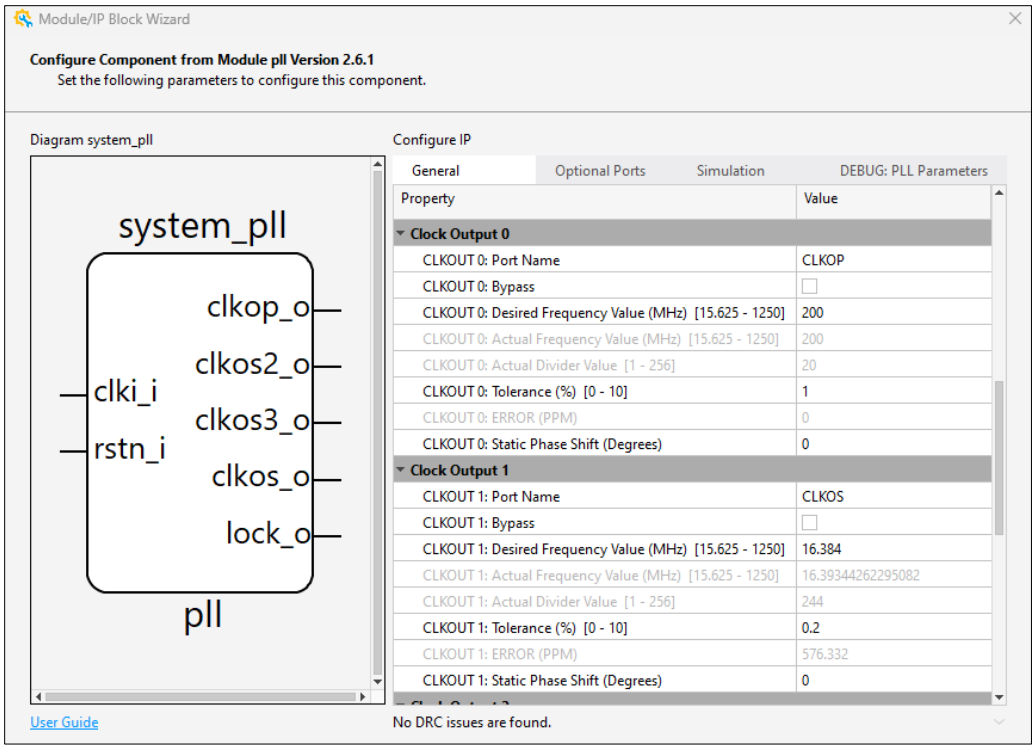


Figure 2.35. PLL IP Configuration – General for system_pll

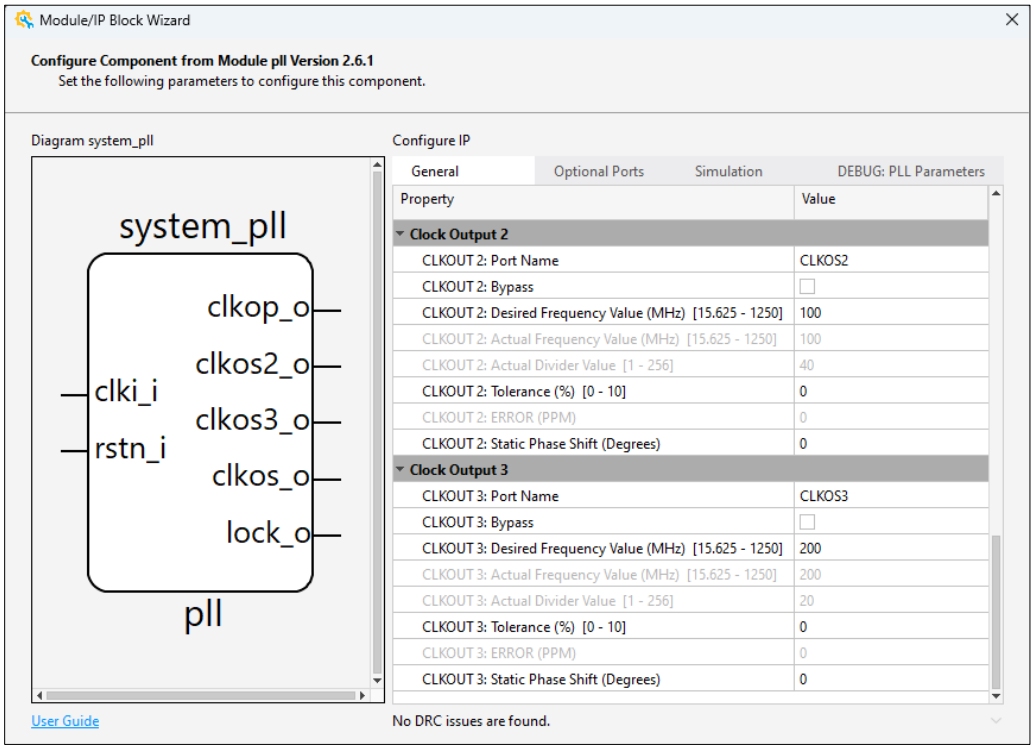


Figure 2.36. PLL IP Configuration – General for system_pll

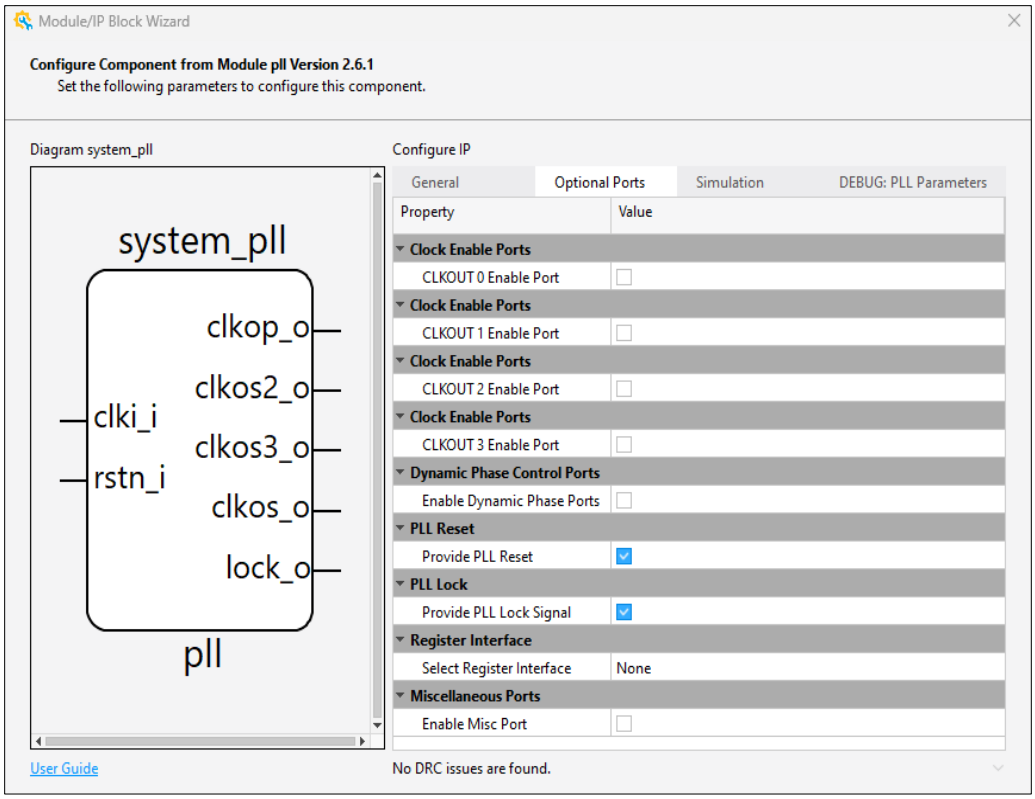


Figure 2.37. PLL IP Configuration – Optional Ports for system_pll

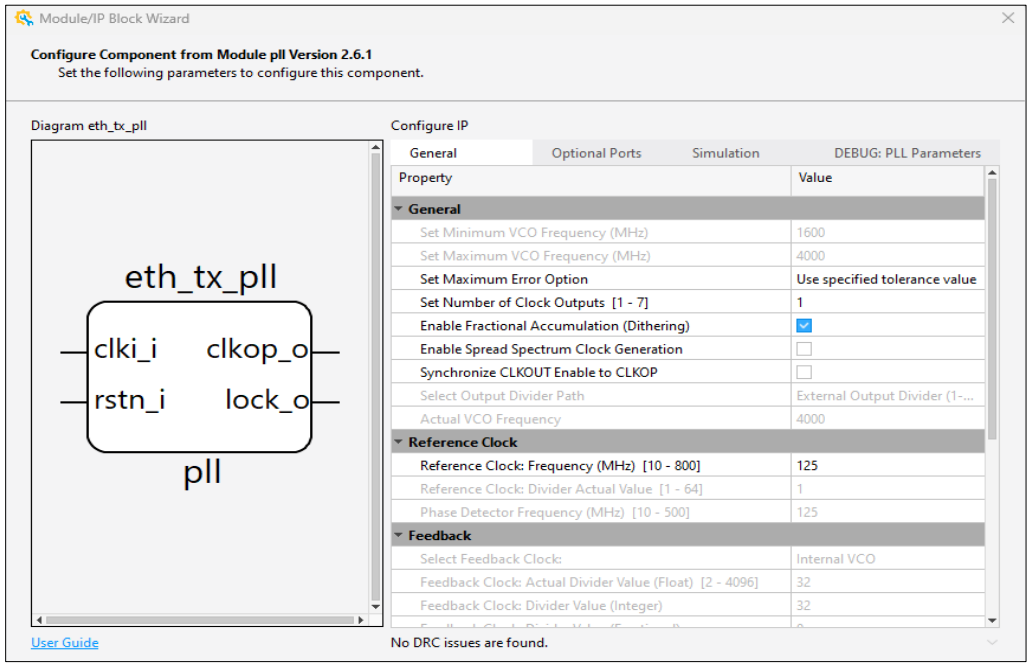


Figure 2.38. PLL IP Configuration – General for eth_tx_pll

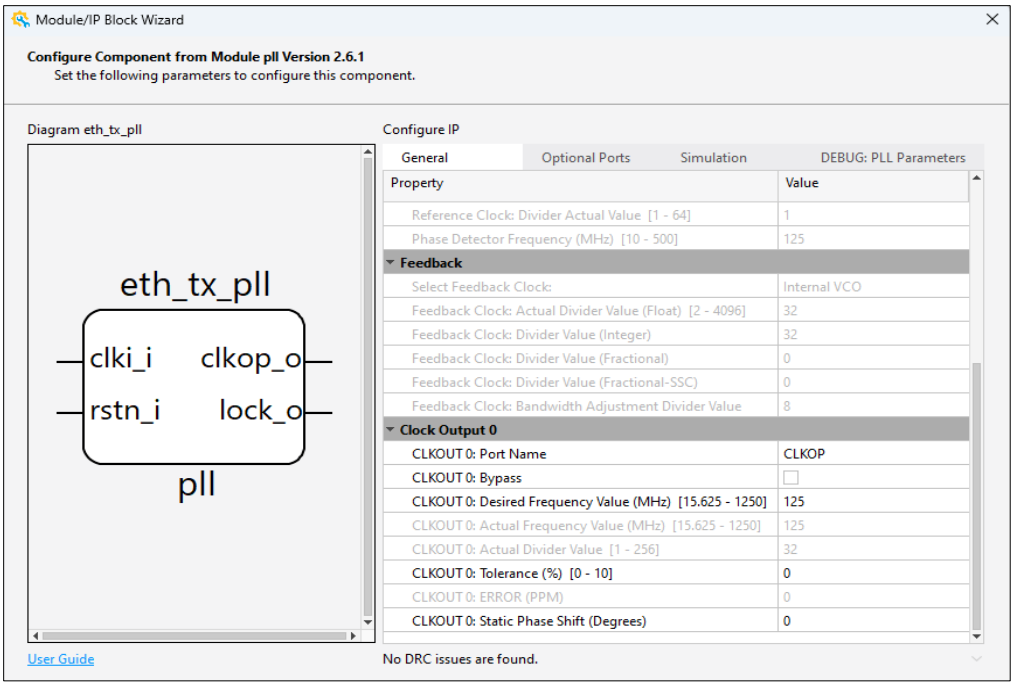


Figure 2.39. PLL IP Configuration – General for eth_tx_pll

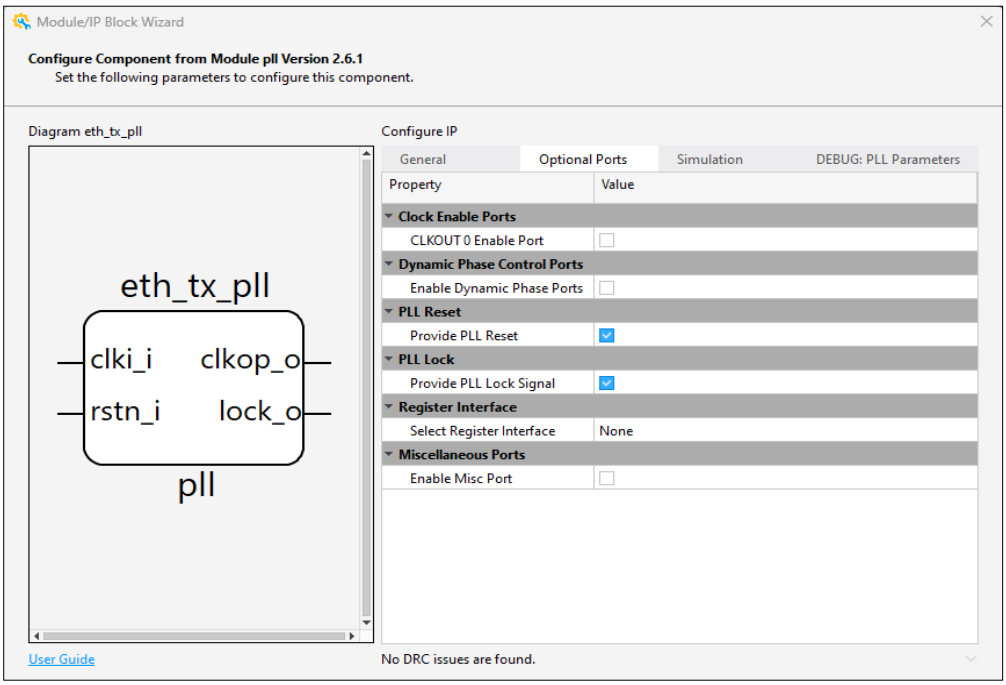


Figure 2.40. PLL IP Configuration – General for eth_tx_pll

2.5.14. Reset Modules

A reset synchronizer module ensures that an asynchronous reset signal is safely brought into a synchronous clock domain without causing metastability or glitches on reset signals.

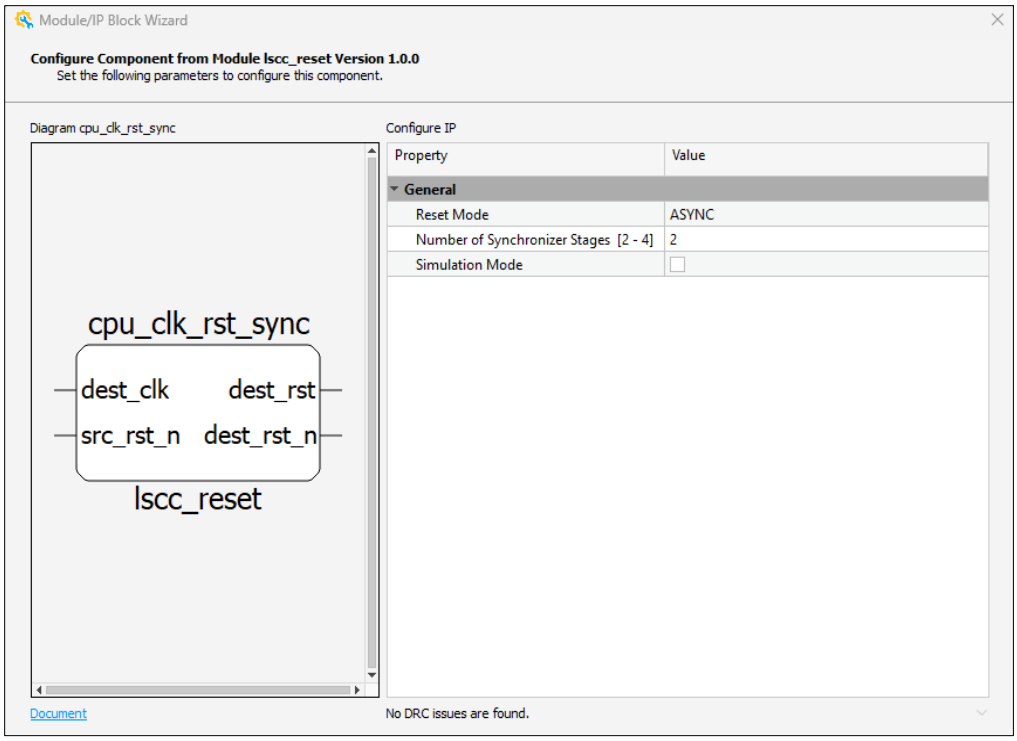


Figure 2.41. Reset Module Configuration – General

2.5.15. AXIS FIFO Module

The AXIS FIFO IP is used to buffer data between two different clock domains while maintaining the AXI stream protocol.

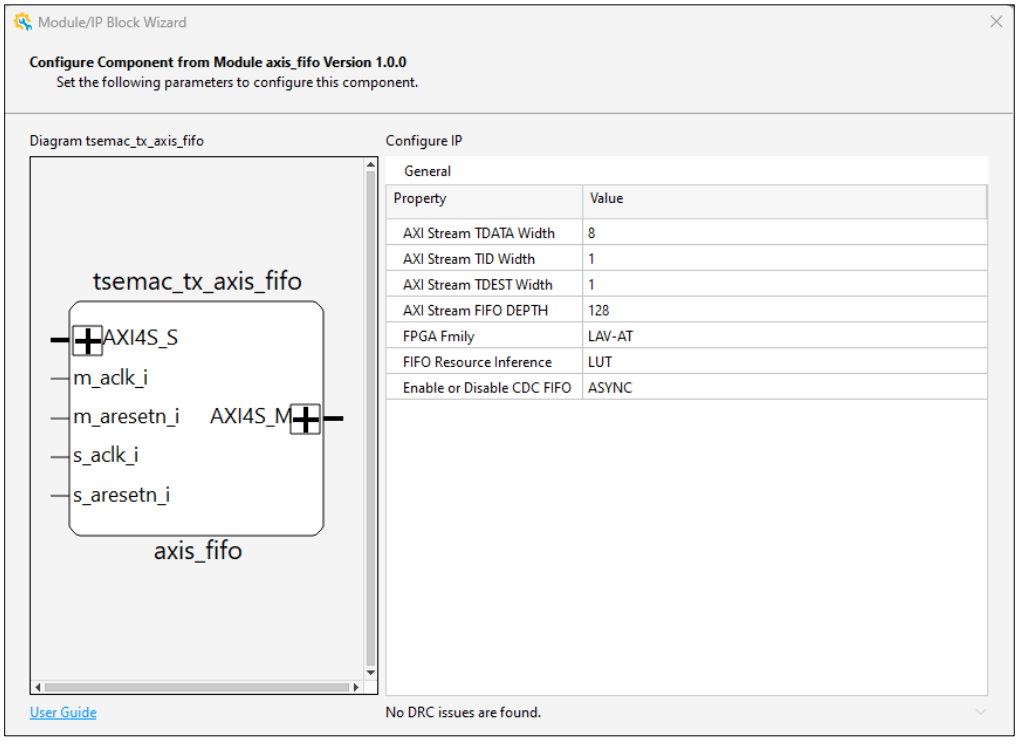


Figure 2.42. AXIS FIFO Module Configuration – General for tsemac_tx_axis_fifo

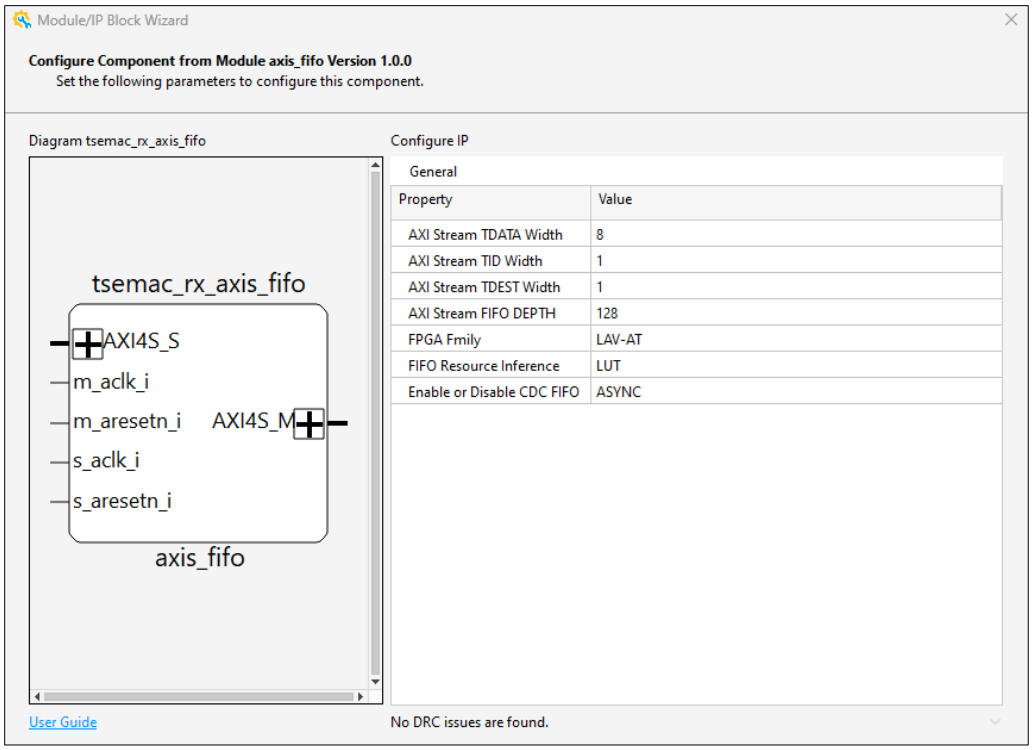


Figure 2.43. AXIS FIFO Module Configuration – General for tsemac_rx_axis_fifo

2.6. System Level Interfaces

Table 2.5. System Level Interfaces

Top-Level Interface Name	Supported Protocol	Description
Advanced eXtensible Interface 4	AXI4	Used for Data/Control Interfaces on all IPs
Advanced Peripheral Interface	APB	Used as Control Interface for low-speed IP and DDR Memory Controller
Serial Peripheral Interface	SPI	Used for communication with external SPI Flash
Dual Data Rate	LPDDR4	Used for communication with external LPDDR4 SDRAM
1 Gbps Ethernet	RGMII	Used for communication with external Ethernet PHY on FMC daughter card
Management Data Input/Output	MDIO	Used for configuring the external Ethernet PHY on FMC daughter card
Universal Asynchronous Receiver/Transmitter	UART	Used for CPU printouts
General Purpose I/O	GPIO	Used for LED to indicate the RGMII link speed

2.7. SoC Memory/Address Map

Table 2.6. Address Map of GHRD

Base Address	End Address	Size (kB/MB/GB)	Subordinate
0x8000_0000	0x8001_FFFF	256 kB	System Memory
0x0000_0000	0x7FFF_FFFF	2 GB	DDR Memory Controller AXI
0xC000_0000	0xC000_0FFF	4 kB	UART APB
0xC000_1000	0xC000_1FFF	4 kB	GPIO APB
0xC000_2000	0xC0002_FFF	4 kB	Multi-Boot Config APB
0xC000_3000	0xC000_3FFF	4 kB	SGDMA Controller APB
0xC000_4000	0xC000_7FFF	16 kB	TSE IP APB
0xC000_8000	0xC000_AFFF	12 kB	Reserved
0xC000_B000	0xC000_BFFF	4 kB	DDR Memory Controller APB
0xC000_D000	0xC3FF_FFFF	—	Reserved
0xC400_0000	0xC400_0FFF	4 kB	Octal SPI Controller
0xC400_1000	0xF1FF_FFFF	—	Reserved
CPU Local Memory			
0xF200_0000	0xF20F_FFFF	1 MB	CLINT
0xFC00_0000	0xFC3F_FFFF	4 MB	PLIC
0xF000_0000	0xF000_03FF	1 kB	Reserved_Space1
0xF000_0400	0xF1FF_FFFF	31 MB	Reserved_Space2
0xF210_0000	0xFBFF_FFFF	159 MB	Reserved_Space3
0xFC40_0000	0xFFFF_FFFF	60 MB	Reserved_Space4

2.8. Design Constraints

The design constraints are divided into two parts, Pre-Synthesis (SDC) and Post-Synthesis Physical (PDC) constraints. They are used to ensure that the design meets the required performance, timing closure, functionality and physical placement requirements as per the FPGA device.

Table 2.7. Design Constraints

Sr No	Constraint Type	File name	Comment
1	Clock	constraints.sdc	Lists all generated clock, created clock used by design.
2	Delay	<Project_name>.pdc	Lists inter board delay and false path definitions.
3	I/O	<Project_name>.pdc	Pin locking of all I/O and I/O standard matching board requirement.

2.9. Resource Utilization

Figure 2.44 shows the approximate GHRD resource utilization and Table 2.8 shows the total LUT4, PFU register, I/O buffer, and EBR resource utilization.

Map Resource Usage

impl_1	LUT4	Logic	Distributed RAM	Ripple Logic	PFU Registers	IO Registers	IO Buffers	DSP MULT	EBR
soc_golden_gsrd	31474(0)	23332(0)	4776(0)	3366(0)	24408(94)	2(0)	95(30)	4(0)	106(0)
tsemac_tx_axis_fifo_inst	217(0)	85(0)	120(0)	12(0)	81(0)	0(0)	0(0)	0(0)	0(0)
tsemac_rx_axis_fifo_inst	224(0)	92(0)	120(0)	12(0)	81(0)	0(0)	0(0)	0(0)	0(0)
system_uart_inst	661(0)	611(0)	0(0)	50(0)	608(0)	1(0)	0(0)	0(0)	0(0)
system_pll_inst	25(0)	25(0)	0(0)	0(0)	17(0)	0(0)	0(0)	0(0)	0(0)
system_ospi_fc_inst	2076(0)	1682(0)	192(0)	202(0)	1668(0)	0(0)	4(0)	0(0)	2(0)
system_lpddr4_mc_inst	8647(0)	6565(0)	1152(0)	930(0)	7303(0)	0(0)	49(0)	0(0)	25(0)
system_ic_inst	8586(1)	5158(1)	3186(0)	242(0)	5666(0)	0(0)	0(0)	0(0)	0(0)
system_gpio_inst	126(0)	126(0)	0(0)	0(0)	97(0)	0(0)	8(0)	0(0)	0(0)
system_boot_mem_inst	1290(0)	1214(0)	0(0)	76(0)	669(0)	0(0)	0(0)	0(0)	64(0)
system_apb_ic_inst	133(0)	133(0)	0(0)	0(0)	7(0)	0(0)	0(0)	0(0)	0(0)
sys_clk_rst_sync_inst	0(0)	0(0)	0(0)	0(0)	2(0)	0(0)	0(0)	0(0)	0(0)
riscv_cpu_rx_inst	4805(0)	4017(0)	6(0)	782(0)	3343(0)	0(0)	0(0)	4(0)	7(0)
perip_clk_rst_sync_inst	0(0)	0(0)	0(0)	0(0)	2(0)	0(0)	0(0)	0(0)	0(0)
mac_txclk_rst_sync_inst	0(0)	0(0)	0(0)	0(0)	2(0)	0(0)	0(0)	0(0)	0(0)
mac_rxclk_rst_sync_inst	0(0)	0(0)	0(0)	0(0)	2(0)	0(0)	0(0)	0(0)	0(0)
lpddr4_sclk_rst_sync_inst	0(0)	0(0)	0(0)	0(0)	2(0)	0(0)	0(0)	0(0)	0(0)
jtaghub_top_inst	39(38)	27(26)	0(0)	12(12)	71(71)	0(0)	4(4)	0(0)	0(0)
eth_tx_pll_inst	23(0)	23(0)	0(0)	0(0)	17(0)	0(0)	0(0)	0(0)	0(0)
eth_tse_mac_inst	2092(0)	1662(0)	0(0)	430(0)	2395(0)	0(0)	0(0)	0(0)	4(0)
eth_sgdma_inst	2181(0)	1617(0)	0(0)	564(0)	2007(0)	0(0)	0(0)	0(0)	4(0)
eth_osc_mdc_inst	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
dual_boot_fpga_config_inst	59(0)	53(0)	0(0)	6(0)	69(0)	1(0)	0(0)	0(0)	0(0)
cpu_clk_rst_sync_inst	0(0)	0(0)	0(0)	0(0)	2(0)	0(0)	0(0)	0(0)	0(0)
axi_clk_rst_sync_inst	0(0)	0(0)	0(0)	0(0)	2(0)	0(0)	0(0)	0(0)	0(0)
axi4_to_apb_bridge_inst	290(0)	242(0)	0(0)	48(0)	201(0)	0(0)	0(0)	0(0)	0(0)

Figure 2.44. GHRD Approximate Resource Utilization

Table 2.8. GSRD Total Approximate Resource Utilization

Resource	Approximate Usage	Approximate Percentage Utilization
LUT4 (Logic + Distributed RAM + Ripple Logic)	31474	7.919%
PFU Register	24408	6.141%
I/O Buffers	95	17.056%
EBR	106	10.707%

3. Signal Description

Table 3.1 shows the input/output interface signals for the top-level module.

Table 3.1. Top-level I/O

Signal Name	I/O Type	I/O Width	Description
system_pll_clk_i	Input	1	Reference clock input for internal PLLs
lpddr4_pll_refclk_i	Input	1	Reference clock input for DDR Memory Controller internal PLL
system_rstn_i	Input	1	Active low reset input for design. Activate by pressing SW1 push button on the board.
GPIO			
gpio_o	Input/Output	8	General Purpose I/O signals connect to LEDs on board. gpio_o[1:0] (LED D23:D22) 00, 01 and 11: Not defined 10: 1000 Mbps. D23 = ON, D22 = OFF gpio_o[7:2] are not used
UART			
uart_txd_o	Output	1	UART transmits output. Connects to the board TXD signal.
uart_rxd_i	Input	1	UART receives input. Connects to the board RXD signal.
Octal SPI Controller			
ospi_clk	Output	1	Serial clock to SPI Flash
ospi_ss_n_o	Output	1	SPI Flash chip selects
ospi_dt_io	Input/Output	4	Serial data between FPGA and external SPI Flash
LPDDR4 Memory Controller			
lpddr4_ca_o	Output	6	LPDDR4 command/address
lpddr4_ck_o	Output	1	LPDDR4 clock
lpddr4_cke_o	Output	1	LPDDR4 clock enable
lpddr4_cs_o	Output	1	LPDDR4 chip select
lpddr4_dmi_io	Input/Output	4	LPDDR4 data mask
lpddr4_dq_io	Input/Output	32	LPDDR4 Data
lpddr4_dqs_io	Input/Output	4	LPDDR4 data strobe
lpddr4_reset_n_o	Output	1	External Memory chip reset signal
lpddr4_init_done_o	Output	1	Connects to LED D12 for LPDDR4 initialization status check 0: Initialization and training are completed. LED = ON 1: Initialization and training are in progress. LED = OFF
lpddr4_pll_lock_o	Output	1	Connects to LED D13 for LPDDR4 PLL lock status check 0: PLL is locked. LED = ON 1: PLL is unlocked. LED = OFF
lpddr4_trn_err_o	Output	1	Connects to LED D11 LPDDR4 training status check 0: Training fails. LED = ON 1: Training passes. LED = OFF
RGMII			
rgmii_rxd_i	Input	4	RGMII receive data from Ethernet PHY
rgmii_rxctl_i	Input	1	RGMII receive control from Ethernet PHY
rgmii_rxc_i	Input	1	RGMII receive clock from Ethernet PHY
rgmii_txd_o	Output	4	RGMII transmit data to Ethernet PHY
rgmii_txctl_i	Output	1	RGMII transmit control to Ethernet PHY
rgmii_txc_i	Output	1	RGMII transmit clock to Ethernet PHY
phy_resetn_o	Output	1	Ethernet PHY resetn
tx_pll_clk_en_o	Output	1	FMC daughter card oscillator enable/disable

Signal Name	I/O Type	I/O Width	Description
MDIO			
rgmii_mdc	Output	1	MDIO Clock
rgmii_mdio	Input/Output	1	MDIO Data
Multi-Boot			
config_active_o	Output	1	Connects to LED D10 to check Multi-Boot Configuration block status 0: Configuration is in progress. LED = ON 1: Configuration is done. LED = OFF

4. Software Components

The GSRD (Golden System Reference Design) is enabled by RISC-V core based on FreeRTOS and Lattice FPGA IP drivers. Lattice developed BSP (Board Support Package) drivers in C language act as intermediaries, facilitating communication between the hardware elements on the FPGA and FreeRTOS software. During boot up, these drivers initialize and configure FPGA peripherals to establish effective coordination with the RISC-V processor.

Lattice GSRD uses the open-source lightweight TCP/IP (lwIP) Ethernet stack. It supports ICMP ping in lwIP TCP/IP Ethernet stack. This example demonstrates how GSRD receives an ICMP (Internet Control Message Protocol) packet Echo Request and sends an Echo Reply to the target which is Windows based or Linux based PC. The IP addresses for both the PC and GSRD are hardcoded to 192.168.1.2 and 192.168.1.4 respectively.

During a ping session, the PC sends out an ICMP echo request packet and wait for an ICMP echo reply from Lattice GSRD. The request packet is sent with a specific timestamp, and upon receiving a valid ICMP echo reply, the PC generates metrics such as packet loss, elapsed time, and other relevant data.

4.1. Primary and Golden Bootloader

Bootloader is a bare-metal program that does the following IP configurations:

- Configures the GPIO and UART IP
- For LPDDR4 Memory Controller configuration, it first reads if the PLL Lock status is set and then initiates Memory Training and waits until training is completed. Configures the Octal SPI Controller to read the application software stored into external flash while FIFO is disabled using API and copies it into LPDDR4 SDRAM. It then calculates the CRC value on the entire application software copied into LPDDR4 SDRAM by using crc16_ccit() API and compares the value with the original CRC value.
- Failure of the CRC check on the Primary application software triggers reconfiguration of the FPGA with the Golden FPGA bitstream image. This is done by triggering Multi-Boot Configuration module. This step is only used in Primary GSRD system.

4.2. Primary and Golden Application

The Primary and Golden applications are implemented in FreeRTOS and lwIP with the following functions:

- Executed by RISC-V RX CPU from LPDDR4 SDRAM where it initializes the BSP and Operating System.
- Initialize the TSE IP by setting up the 1000 Mbps Ethernet link speed, MAC address and full duplex mode based on the selected configuration.
- Configure external Ethernet PHY through MDIO interface.
- Configure the lwIP with the IP address (default to 192.168.1.4), and setup the netmask and gateway address for ping connection.
- Configure the SGDMA Controller MM2S (Memory-Mapped to Streaming) interface for transferring the Ethernet ping packets to TSE MAC and set up S2MM (Streaming to Memory Mapped) interface to receive ping packets from the TSE MAC.
- Runs FreeRTOS scheduler and Task Handler. Every incoming Ethernet ping packet is displayed with some minor dots' indicator on the UART terminal.

5. Theory of Operation

The GSRD system comprises two SoCs (System on Chip) designs and corresponding software stacks, called Primary and Golden images. Each SoC design is linked with its respective bootloader and FreeRTOS application software. These SoCs are built using Lattice Propel Builder, and Lattice Radiant software tools. The Lattice Propel SDK provides a software development environment for firmware development. The FPGA image comprises the entire system and is generated by the Radiant tool in the (.bit) bitstream format. The bootloader code is stored inside the System Memory and is part of the bitstream. The bitstream and the FreeRTOS application software are stored in the external SPI Flash (Winbond).

During power-on, the Primary FPGA image is loaded into the device SRAM by the Config Engine. Before the device is completely programmed with the bitstream, the config engine checks the CRC (Cyclic Redundancy Check) for the bitstream to be loaded onto the FPGA. Once the FPGA is configured, the DONE LED on the board glows green. Immediately, the RISC-V starts executing the bootloader software stored inside System Memory and initializes all the soft-IP modules and peripherals to establish a base for communication and data transfers. This process includes configuring the IPs for the desired system operation. After all the IP configuration is complete, RISC-V initiates the Octal SPI Controller to copy the FreeRTOS application software from external SPI Flash into the external LPDDR4 memory device for software execution. Once the application software is copied, the RISC-V checks the CRC on the entire copied application software. If the CRC check fails, RISC-V initiates a soft reset/refresh by instructing the multi-boot configuration module to start the PROGRAMN sequence. The PROGRAMN sequence loads the next bitstream into the FPGA which in this case would be the Golden FPGA and software images, and the same process follows. Once CRC check passes for either Primary or Golden application software, the RISC-V program counter jumps to the application instruction's starting address and starts the FreeRTOS application software execution.

5.1. Boot-Up Sequence

This section describes the RISC-V RX CPU boot up sequence in detail that configures IP drivers, operating modes and bootloader.

- The Golden and Primary application software binaries and their FPGA bitstream images are stored in external SPI Flash before the boot up sequence is initiated.
- The initial bootloader is a part of the internal System Memory ROM embedded into the FPGA bitstream stored into the external SPI Flash. Upon power-on boot up, the Primary bitstream is loaded to the SRAM by Config Engine to configure the FPGA. Then, the bootloader configures the peripherals and GSRD building blocks such as UART, GPIO, LPDDR4 Memory Controller and Octal SPI Controller.
- The bootloader fetches the respective Primary application software through the Octal SPI Controller into System Memory.
- As the application software needs to be executed from external memory, the RISC-V CPU loads the application software into LPDDR4 memory device.
- This application software is stored at the beginning of the LPDDR4 memory mapped address. After the application software is loaded, the RISC-V CPU calculates the CRC of the application code in LPDDR4.
- The CRC of the application software in LPDDR4 memory device is calculated and validated against the reference CRC embedded in SPI flash.
- If the calculated CRC matches the reference CRC, the RISC-V CPU jumps to FreeRTOS execution in LPDDR4 memory device.
- If the calculated CRC mismatches the reference CRC, the RISC-V CPU issues a FPGA REFRESH command to load the Golden bitstream and application software from the SPI Flash.
- Octal SPI Controller performs a self-diagnostic check for read, write and erase operations. Refer to Appendix D. Using Octal SPI Controller for Read, Write, and Erase Self-Diagnostic Check (FreeRTOS Application Software)for more information.
- FreeRTOS initializes SGDMA Controller, TSE IP, and uses MDIO interface of TSE IP to configure external PHY for Ethernet connection.
- LwIP is enabled with the configuration in TSE IP for Ethernet connection.
- FreeRTOS executes the task scheduler.

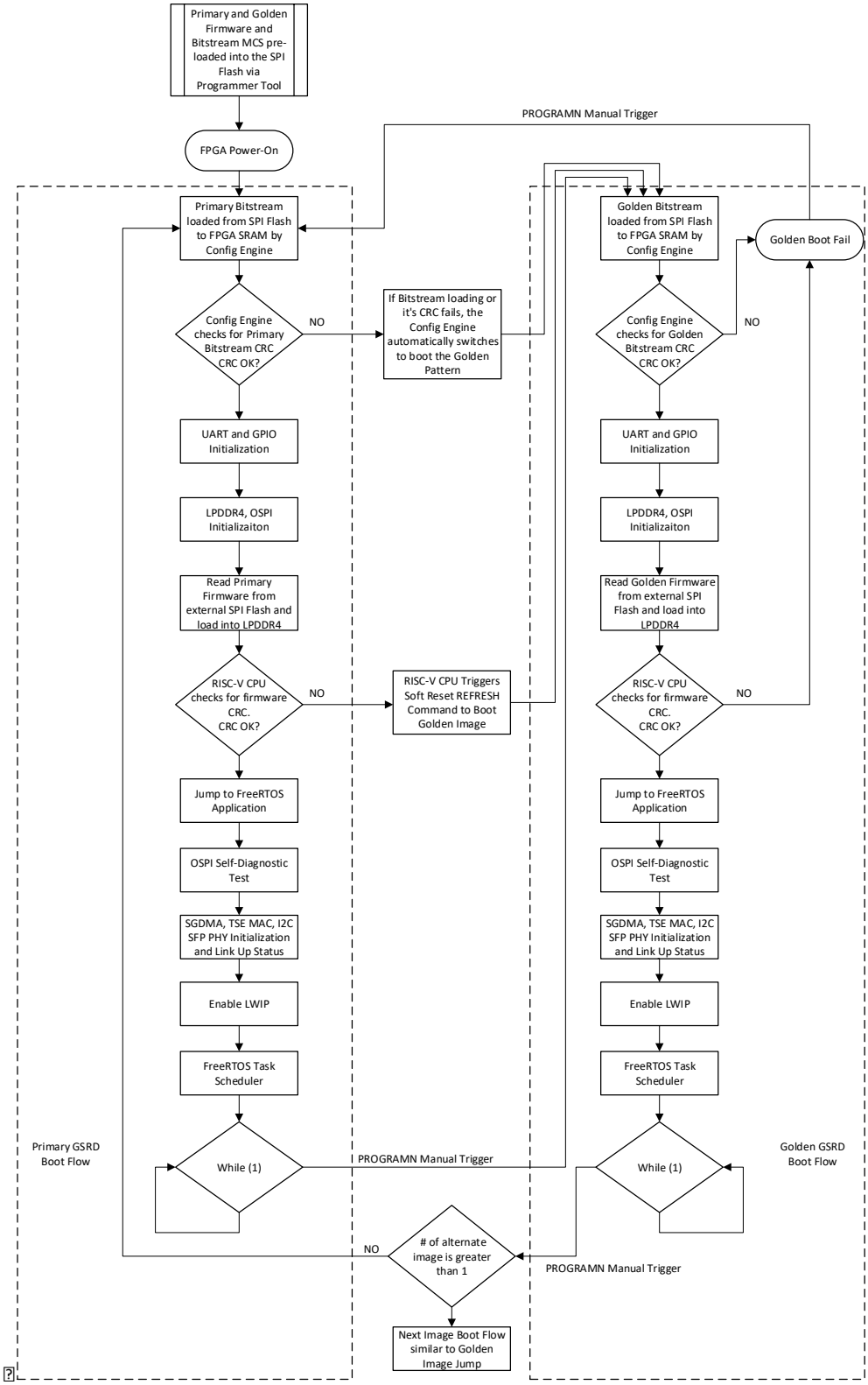


Figure 5.1. GSRD Boot-Up Sequence

5.2. Data Movement

This section describes the RISC-V RX CPU FreeRTOS application software execution in detail.

- Upon the execution of the correct Primary or Golden application software from LPDDR4 SDRAM, the building blocks mentioned earlier are up and running with their associated drivers. For example, if any Ethernet data is expected to arrive, the RISC-V CPU sets up the SGDMA Controller accordingly with receive buffer's address, data length and other configuration modes to route the incoming Ethernet packets.
- When the Ethernet frame is received by the TSE IP, it forwards it to SGDMA Controller to transfer the data to receive buffer in LPDDR4 SDRAM.
- When data is written into the LPDDR4 SDRAM, SGDMA Controller triggers interrupt to RISC-V CPU to acknowledge the data transfer is completed. RISC-V CPU then clears the SGDMA Controller's interrupt status.
- Once interrupt is serviced, the RISC-V goes back to the main task scheduler to execute the next task.

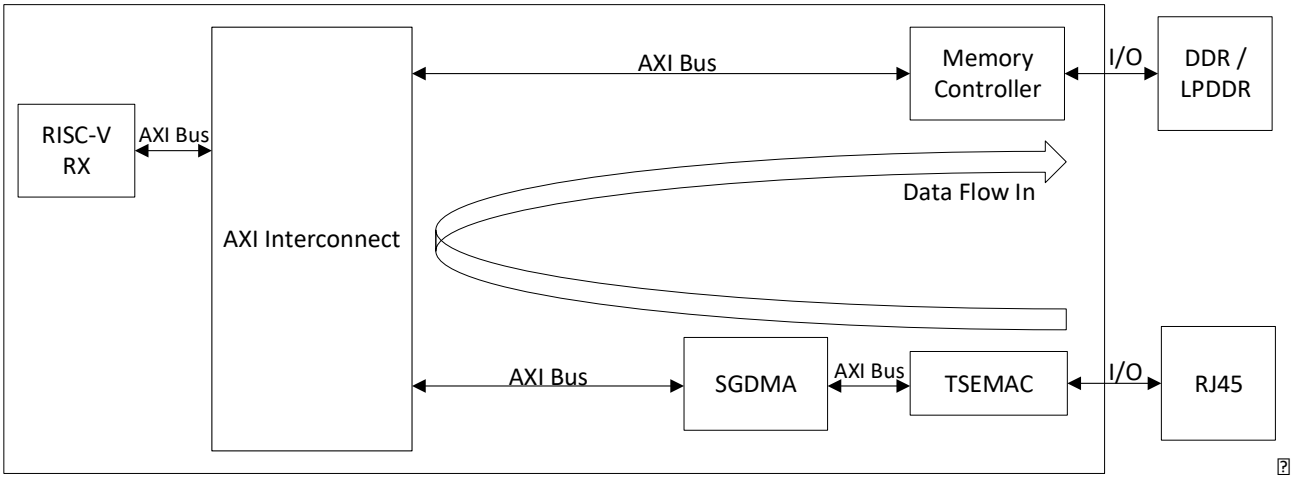


Figure 5.2. Ethernet Data RX Flow

- For outgoing data, data is fetched from a location inside LPDDR4 SDRAM by RISC-V CPU. The SGDMA Controller transfers the data to TSE IP for transmission outside the FPGA.
- When no data activity occurs over Ethernet, the RISC-V CPU continues running idle task in FreeRTOS.

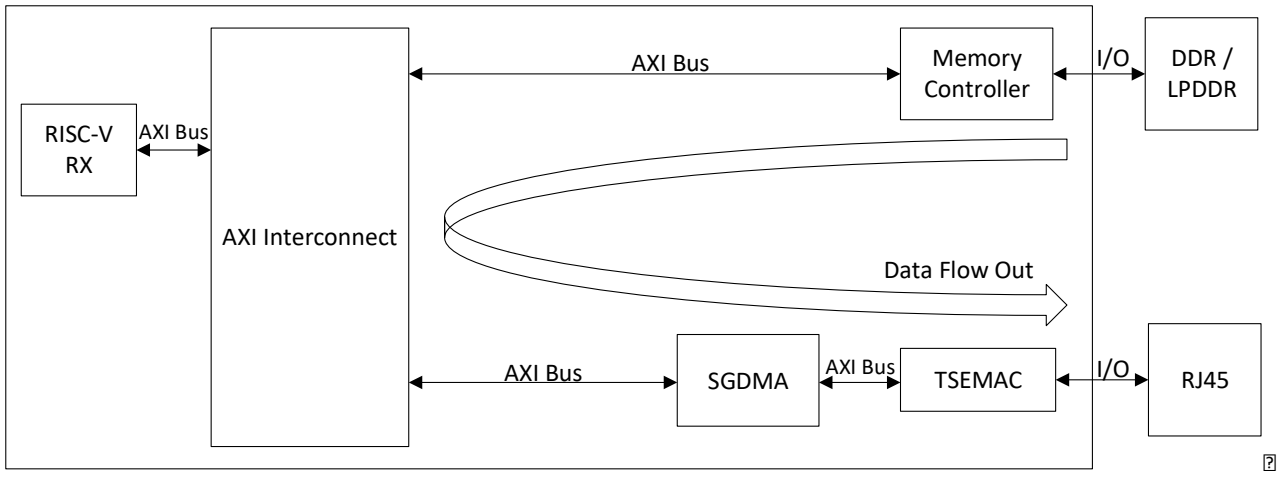


Figure 5.3. Ethernet Data TX Flow

6. Running the GSRD Demonstration

This section describes the procedure for running the GSRD demonstration using the pre-built executables and binary files in the design package. You can skip this chapter if you do not want to run the GSRD demonstration or reference design on hardware.

6.1. Executables

This section provides the directory structure, file names and locations of the executables (SPI Flash) required for running the GSRD demonstration. Your Avant-E Evaluation Board has Winbond flash device as stated in Table 6.1.

Table 6.1. Supported Flash Devices

Board	Flash Device
Avant-E Evaluation Board	Winbond W25Q512JV

- Download the design package from the Lattice Semiconductor website. Go to *Design File* in the GHRD/GSRD Demonstration, download the *Avant-E Golden System Reference Design and Demo V2.0 – Bitstream* file.
- Unzip the .zip file to your local directory, for example to *<C:\user_workspace>.*
- The extracted directory has the following executables listed in Table 6.2.

Below are the bitstreams and software image binaries for programming the FPGA.

Table 6.2. Executable Files for Winbond Flash

File Description	File Name	Starting Address in SPI Flash
Primary Software with CRC	c_primary_appcrc.bin	0x028A 0000
Primary FPGA Bitstream	soc_primary_gsrd_impl_1.bit	0x0000 0000
Golden Software with CRC	c_golden_appcrc.bin	0x0280 0000
Golden FPGA Bitstream	soc_golden_gsrd_impl_1.bit	0x0000 0000
Multi-Boot MCS File (Golden + Primary Bitstream)	multiboot_system.mcs	0x0000 0000

6.2. Setting Up the Hardware

This section provides the procedure for setting up the Avant-E Evaluation board for GSRD demonstration as shown in Figure 6.1, Figure 6.2, and Figure 6.3.

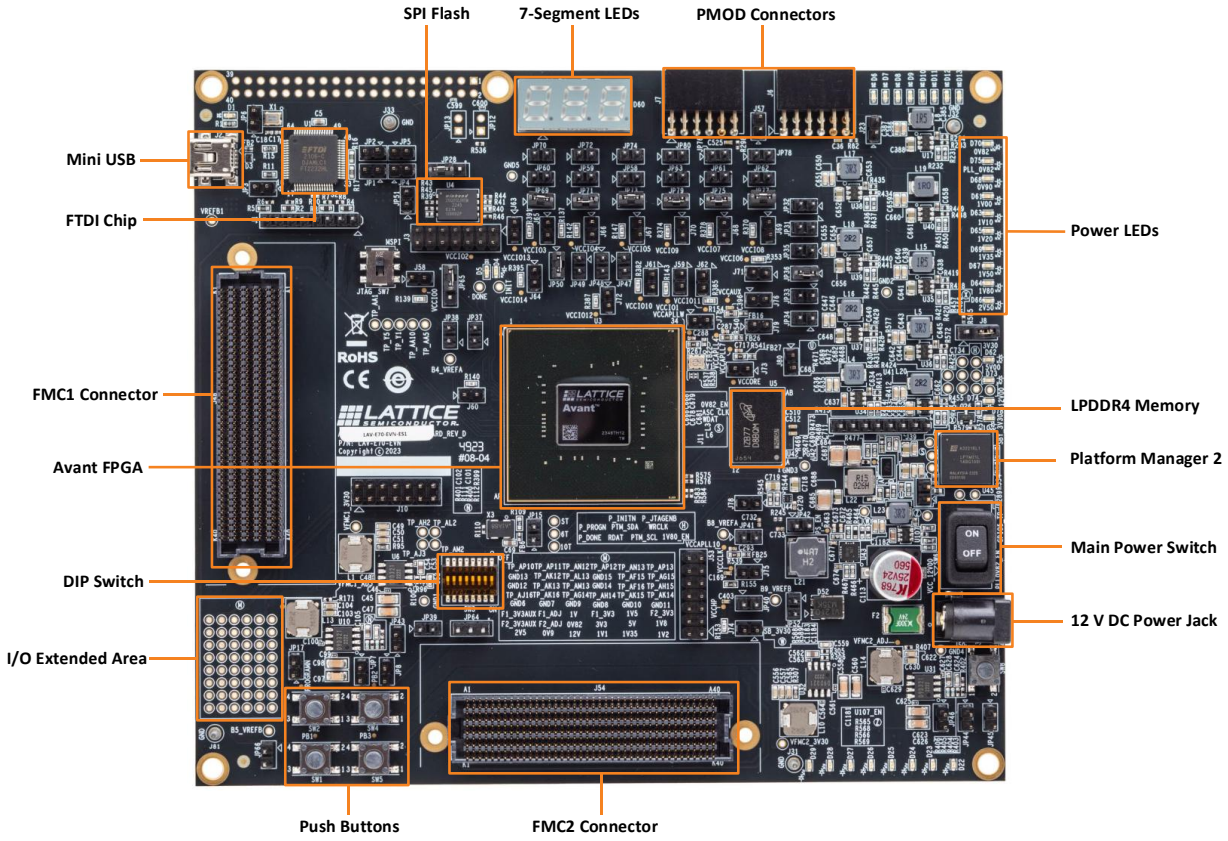


Figure 6.1. Avant-E Evaluation Board

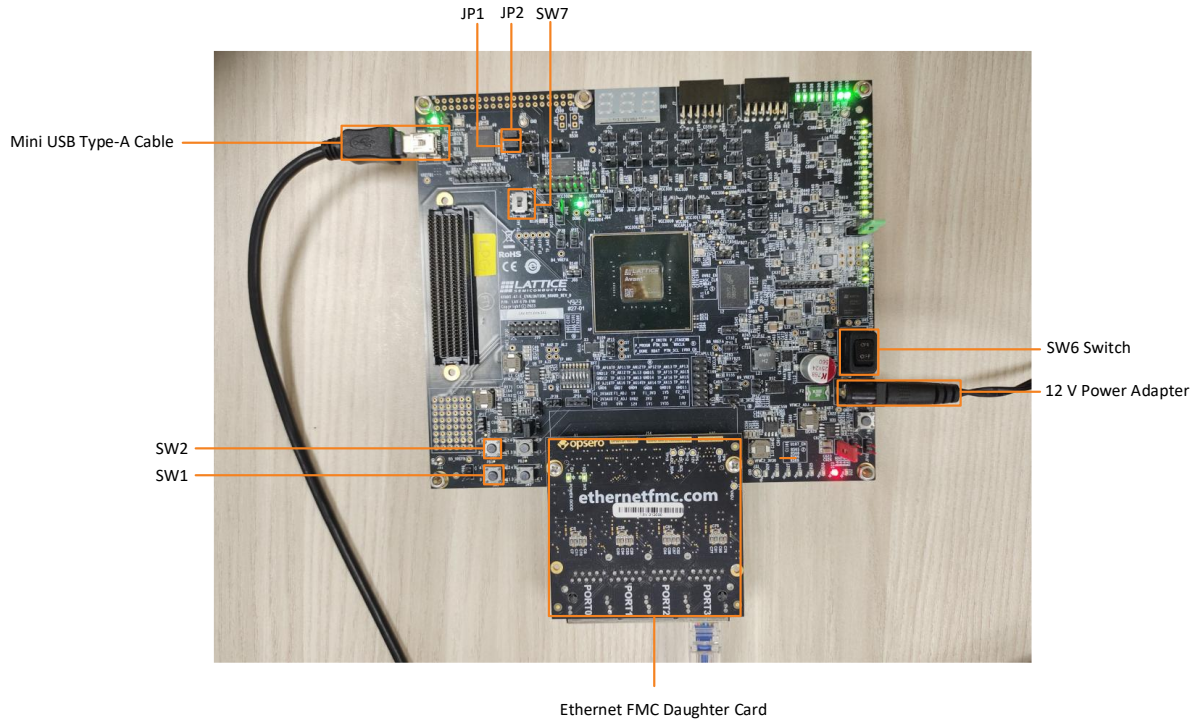


Figure 6.2. Connections, Jumpers and Switches Needed for Demonstration

Figure 6.3. Ethernet PHY FMC Card

To set up the hardware, perform the following:

1. Connect the 12 V power adapter to J50 DC input supply jack.
2. Connect the Mini-USB Type-A cable from PC to J2.
3. Connect jumpers on Pin 1 and Pin 2 on both JP1 and JP2 headers to enable UART.
4. Align and carefully install the Ethernet FMC daughter card onto J54 connector. Secure the card by installing 2 screws to the standoff of the card from the back side of the evaluation board.
5. Connect the Ethernet RJ45 cable from the host PC cable to the Port 3 of FMC daughter card.
6. For executables programming, switch SW7 to JTAG mode.
7. Turn on power switch SW6.

   Notes:
   - SW1 is the FPGA Reset push button to reset the GSRD design.
   - SW2 is the FPGA PROGRAMN push button to switch between Primary and Golden GSRD manually.

6.3. Setting Up the UART Terminal

The software code during the GSRD demonstration displays messages on the terminal through the UART interface.

To set up the UART terminal, perform the following:

1. Connect the Lattice Avant-E Evaluation board to the PC/Laptop using USB Type-A UART cable.
2. Open Propel SDK tool.
3. Double-click on the terminal icon shown in Figure 6.4.

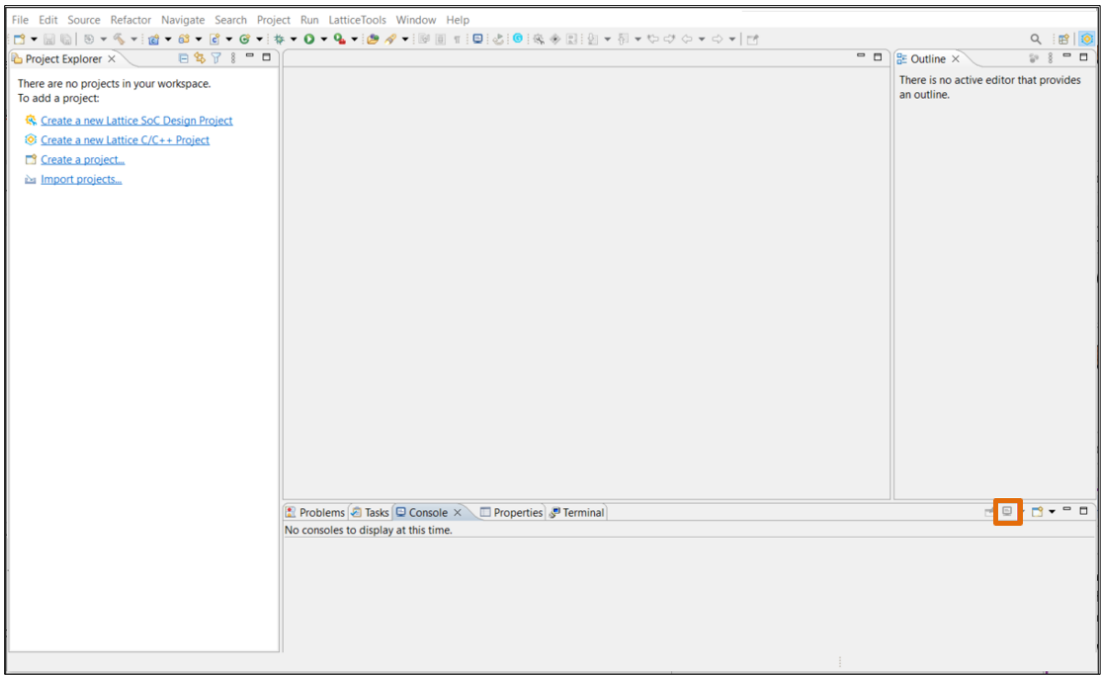


Figure 6.4. UART Terminal Icon on Propel SDK Window

4. Select **Serial Terminal** as shown in Figure 6.5. In Serial port dropdown list, select the last COM in the list as shown in Figure 6.6.
   Note: This detail can also be found under the Ports (COM and LPT) section on your local PC, under Device Manager. The COM port number can be different. If a USB port does not work, try a different USB port.

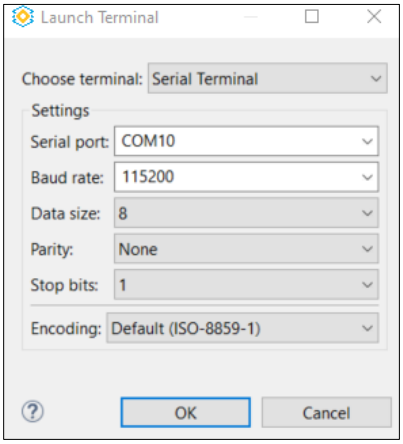


Figure 6.5. UART Launch Terminal Window

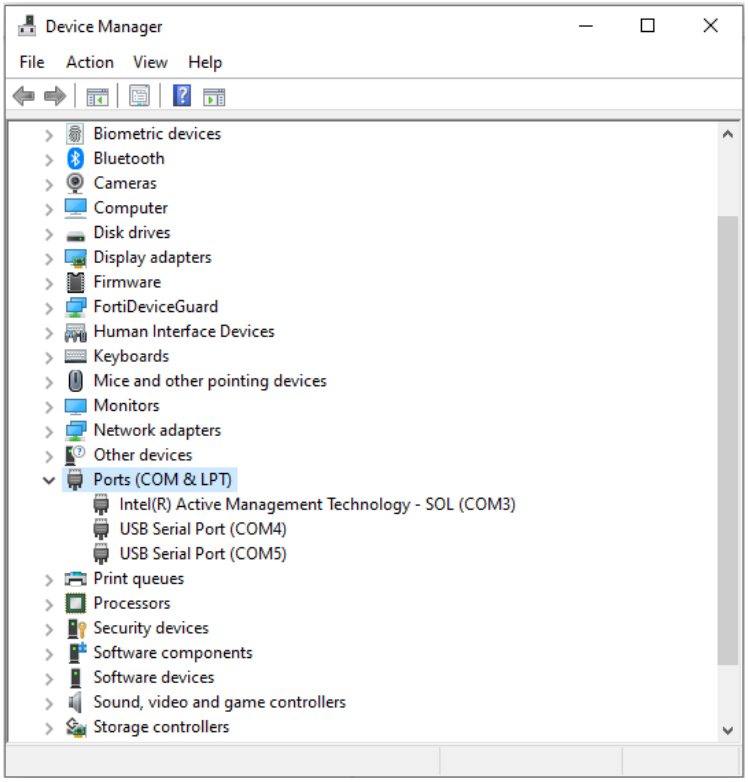


Figure 6.6. Device Manager Window on PC

5. Set Baud rate to **115200**.
6. Click **OK**.

6.4. Setting Up the Non-Volatile Memory Register

For the GSRD design on Lattice Avant-E Evaluation Boards, you need to ensure the settings of the One-Time-Programmable Non-Volatile Configuration Memory. You may skip this section if you have already taken this step before. Otherwise, perform a one-time step by JTAG to modify the default MSPI addressing mode from 24-bit to 32-bit. This is necessary for the multi-boot feature to function properly.

For more information, refer to Appendix A. Changing the SPIM Settings in the NV Register.

6.5. Programming Standalone Golden or Primary GSRD Bitstream and Application Software

To program the standalone golden or primary GSRD bitstream and application software, perform the following:

1. Connect the Avant-E Evaluation Board to a PC using USB cable as per the hardware setup mentioned in Setting Up the Hardware section. Make sure jumpers at JP1 and JP2 are installed properly.
2. Keep SW7 in JTAG mode.
3. Power-on the board.
4. Launch Lattice Programmer tool. In the Getting Started dialog box, select **Create a new blank project**. Browse to the Project Location on your local machine. In this case, it can be the same folder as the downloaded executables folder.

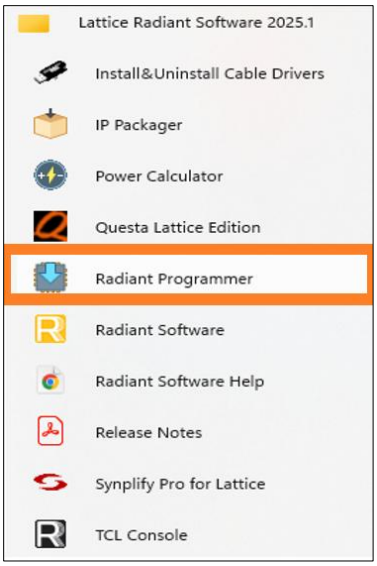


Figure 6.7. Launch Radiant Programmer from Windows Start

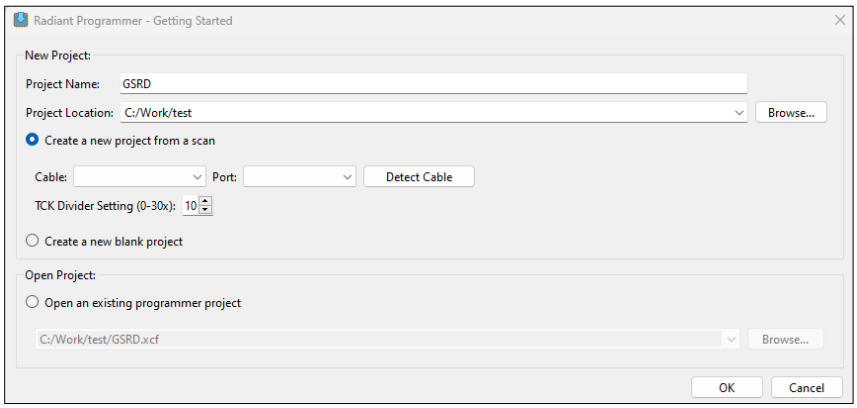


Figure 6.8. Radiant Programmer Start Window

5. Click **OK**.

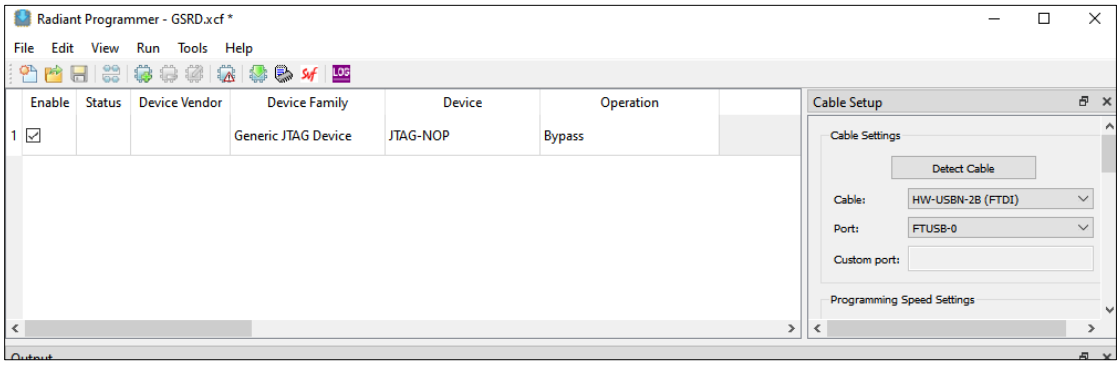


Figure 6.9. Radiant Programmer. xcf Window

6. If the **Device Family** shows as **Generic JTAG Device**, click **Scan Device**, as shown in Figure 6.10, to update the **Device Family** information automatically.

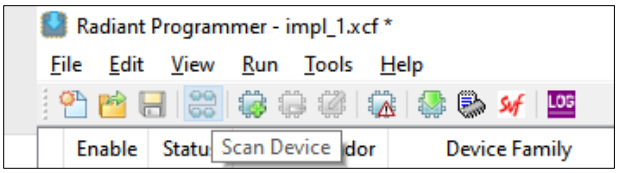


Figure 6.10. Scan Device Icon on Radiant Programmer

Figure 6.11. Select Device for Programming

7. Click on the highlighted item under the **Device** field to un-highlight it as shown in Figure 6.12.

Figure 6.12. Device Selected for Programmer

8. Double-click on the **Operation** tab or right-click and select **Device Properties**.

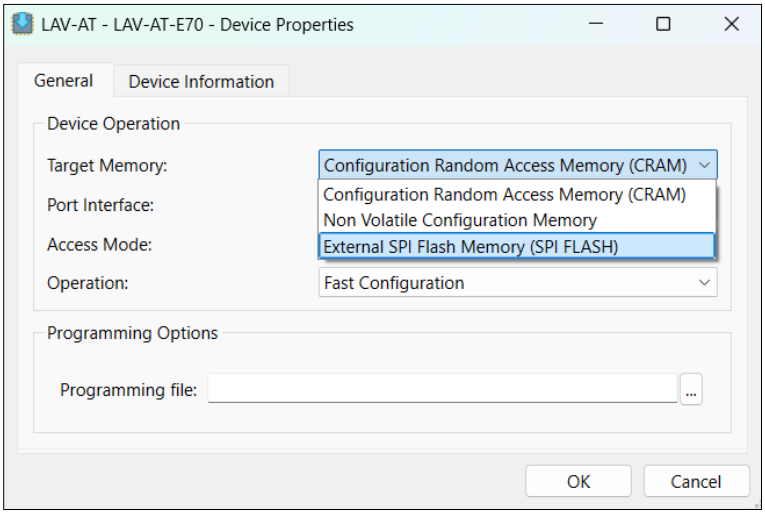


Figure 6.13. Select the Target Memory for Programming – SPI Flash

9. Before programming, you need to erase the entire SPI Flash Memory by applying the settings shown in Figure 6.14.

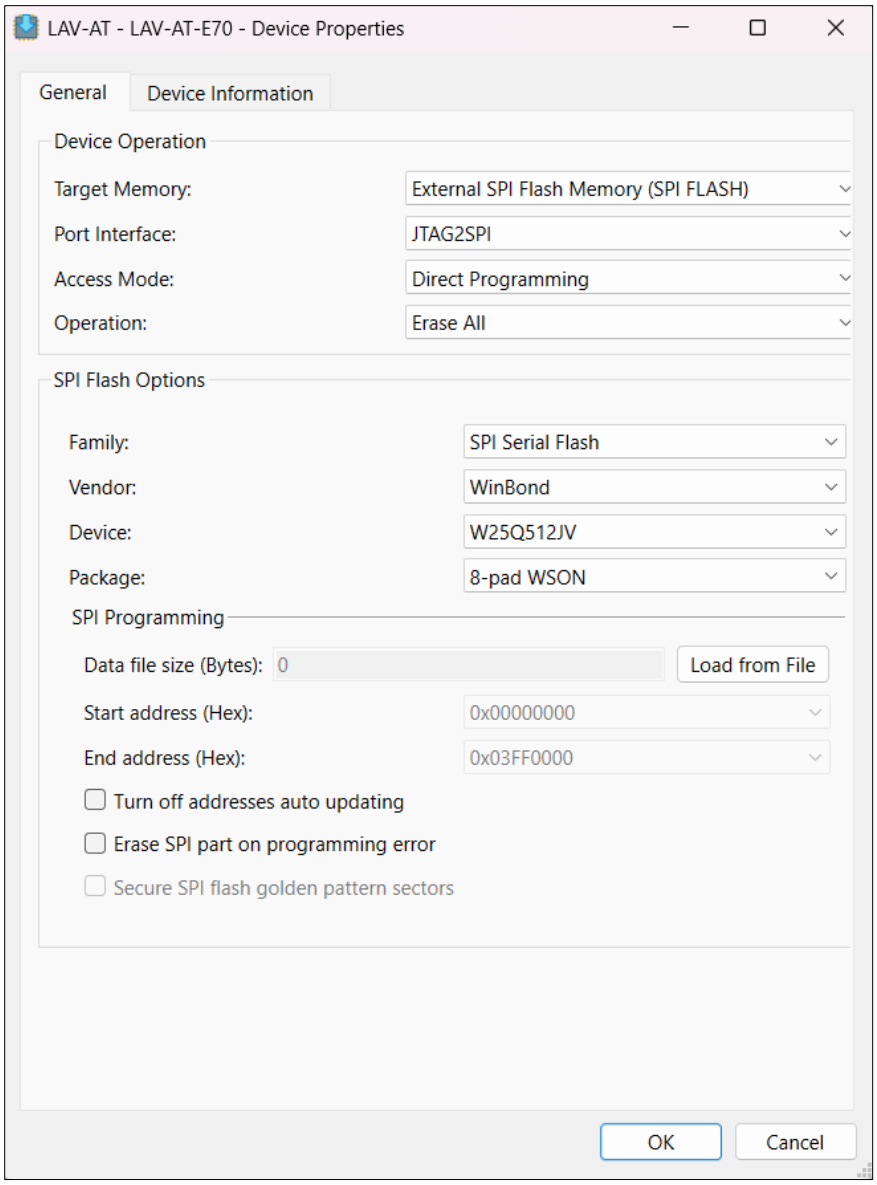


Figure 6.14. Device Properties to Erase the Winbond SPI Flash

10. Click **OK** and click on the Program Device Icon or the menu item, **Run > Program Device**. This erases the entire Flash Memory. Wait for the process to be completed.

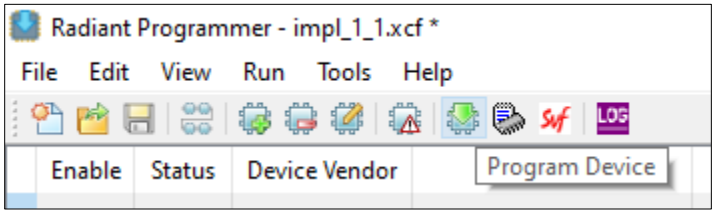


Figure 6.15. Program Button to Program the SPI Flash

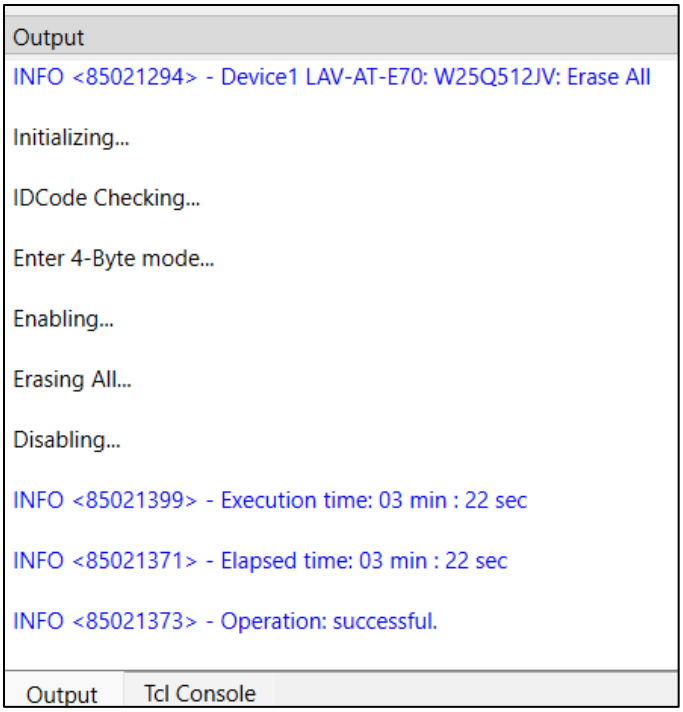


Figure 6.16. Output After Erase All

11. Power cycle the Avant-E Evaluation Board.
12. Erase the FPGA CRAM. Click **OK**.

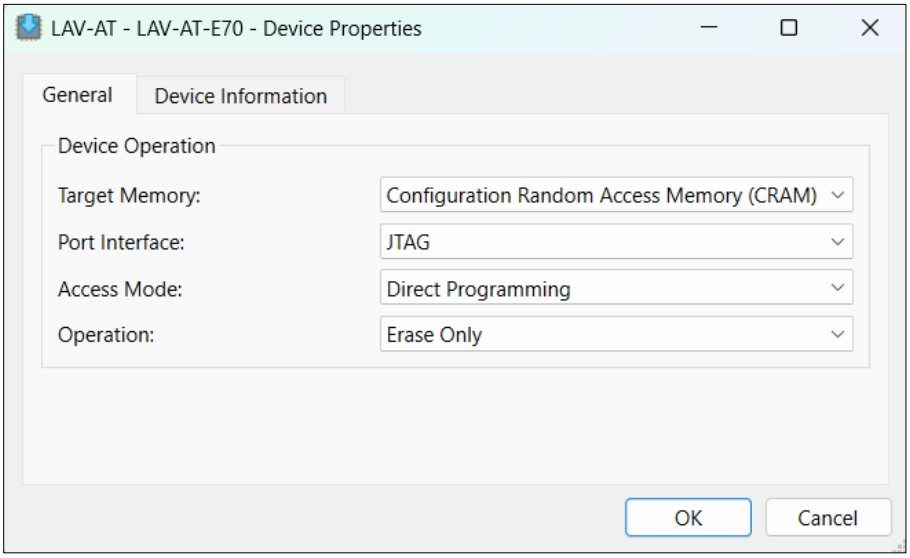


Figure 6.17. Erase Only Operation for CRAM Programming

13. To program the **c_golden_appcrc.bin** file into the SPI Flash, follow the settings as shown in Figure 6.18.

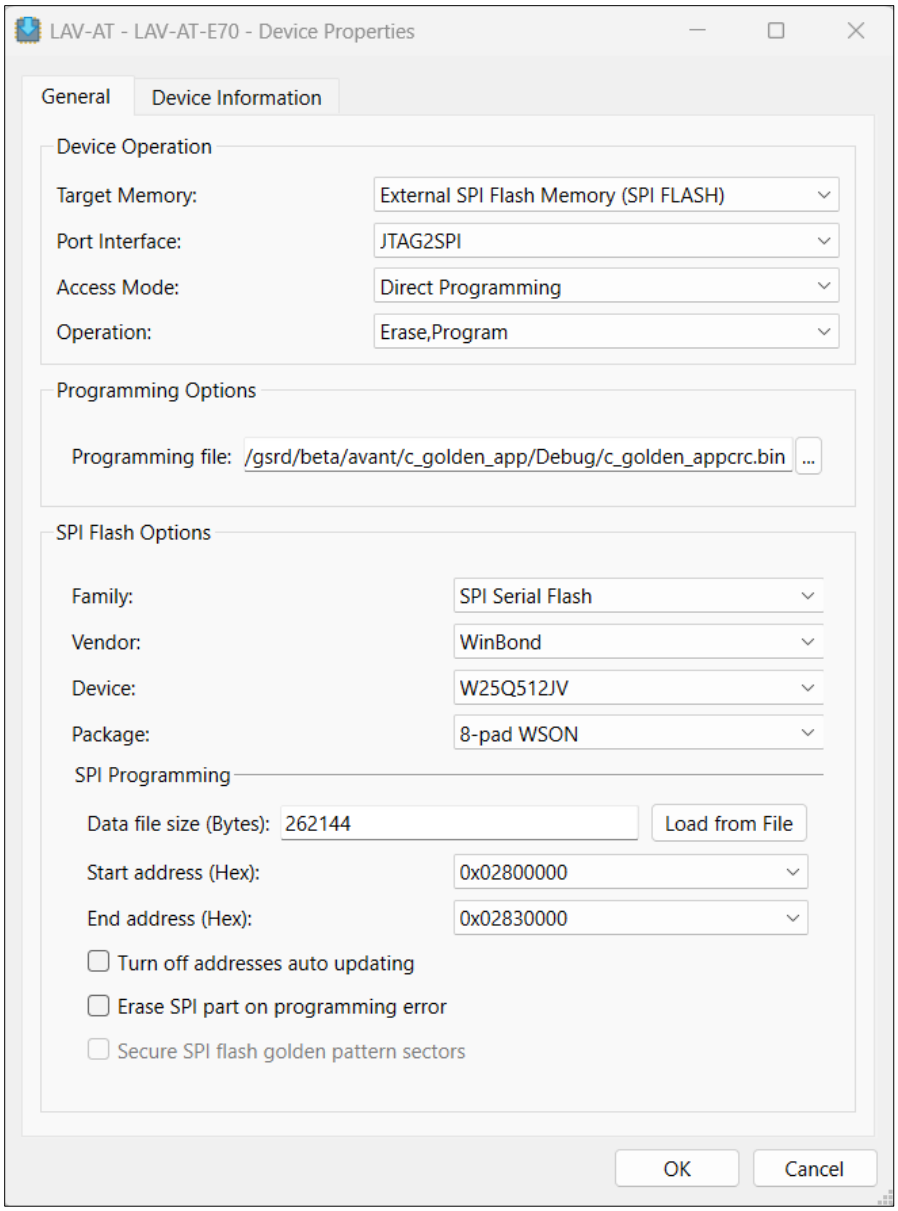


Figure 6.18. Device Properties to Program the Winbond SPI Flash

14. Click **OK**
15. Use TCK Divider Setting (0-30x) to **4**.

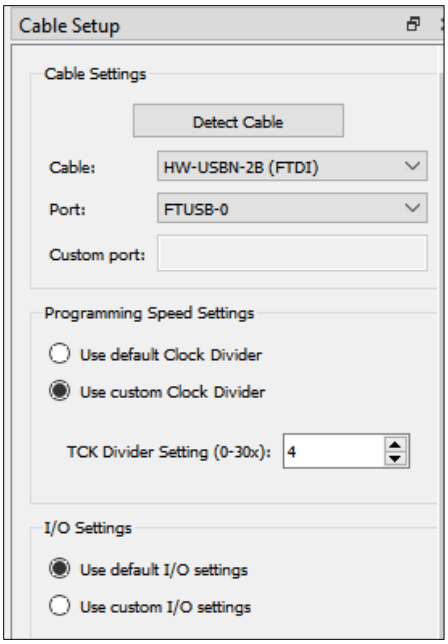


Figure 6.19. Cable Settings for Device Programming

16. Click the **Program Device** Icon or go to the menu item, **Run > Program Device**. The output console displays the Operation Successful message.

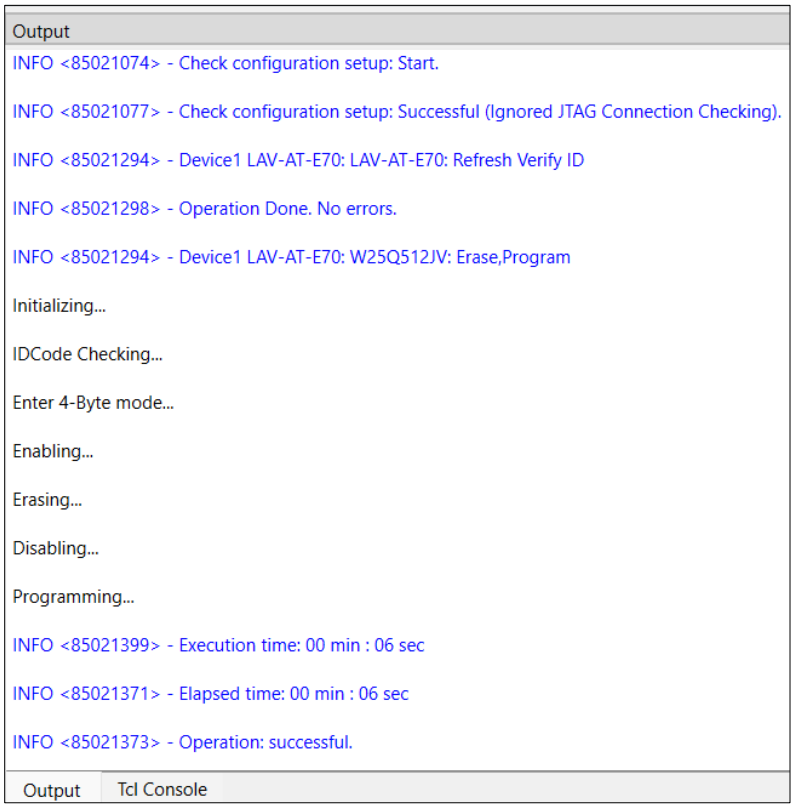


Figure 6.20. Radiant Programmer Console Output after Programming the SPI Flash

17. Power cycle the Avant-E Evaluation Board.

18. To program the FPGA Golden GSRD bitstream, double-click on the **Operation** tab to update the selections as shown in Figure 6.21. Make sure to provide the path to the .bit file location on your local machine where you have unzipped the executables.

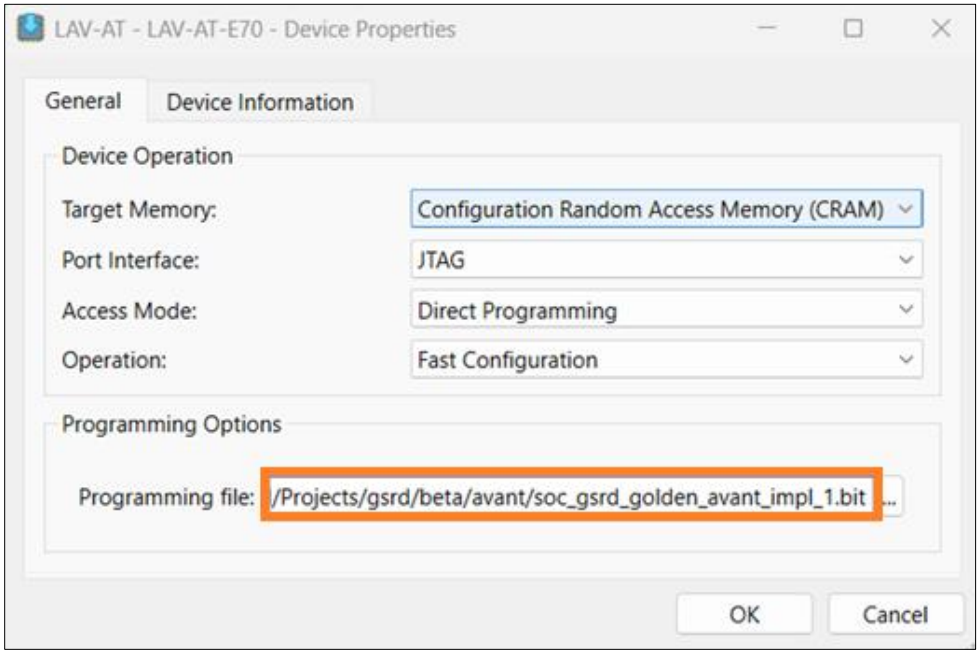


Figure 6.21. Device Properties to Program the FPGA Bitstream in CRAM

19. Click **OK** and click the **Program Device** Icon or go to the menu item**, Run > Program Device**. Wait until the operation is successful as shown in Figure 6.22.

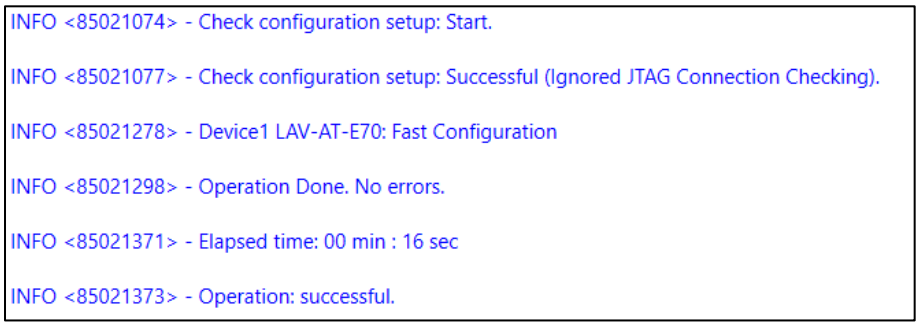


Figure 6.22. Radiant Programmer Console Output after Bitstream is Programmed

20. Set up the UART terminal as mentioned in Setting Up the UART Terminal section.
21. Press **SW1 Reset** button on the board as highlighted in Figure 6.23.

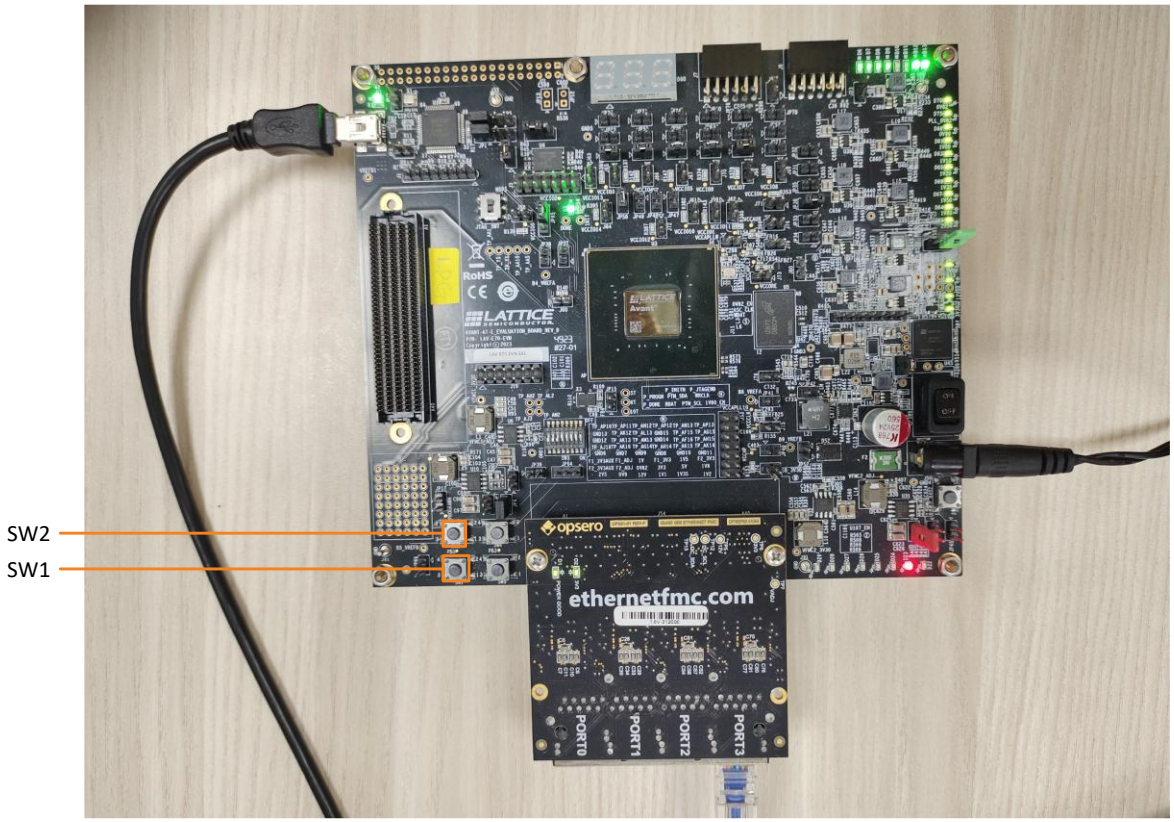


Figure 6.23. SW1 Reset Button and SW2 PROGRAMN Button

22. The results of running the Golden GSRD are as follows:
    - The console should appear with messages as shown in Figure 6.25 and Figure 6.26.
    - Hardware LED Status as shown in Figure 6.24:
        - D23 and D22 – RGMII link speed. D23 = ON, D22 = OFF: 1000 Mbps. Other combinations of LED indication are undefined.
        - D12 – LPDDR4 initialization status. LED = OFF: Initialization and training are in progress. LED = ON: Initialization and training are completed.
        - D13 – LPDDR4 PLL Lock status. LED = OFF: PLL is unlocked. LED = ON: PLL is locked.
        - D11 – LPDDR4 training status. LED = OFF: Training passes. LED = ON: Training fails.
        - D10 – multi-boot configuration status. LED = OFF: Configuration is done. LED = ON: Configuration is in progress.

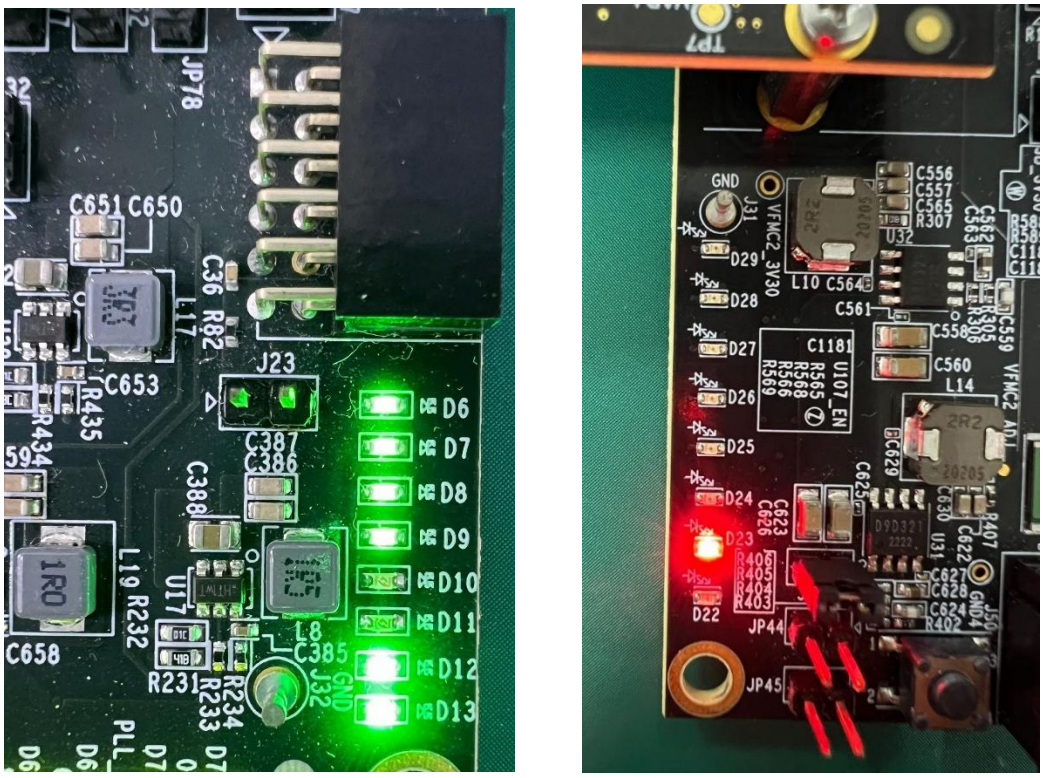


Figure 6.24. LED Status

```
**************************************************
***        GSRD Golden Bootloader Avant-E        ***
**************************************************
Initializing OSPI Controller...Octal SPI Done.

Memory Controller Initialization:
DDR MC training successfully

Memory Controller Initialization Complete.

Reading and verifying firmware CRC value ...
Embedded LPDDR CRC value: 21a9
Calculated Firmware CRC value: 21a9

CRC matches successfully !!

Jumping to FreeRTOS application ...
```

Figure 6.25. Golden GSRD - Output on UART Terminal for Bootloader and FreeRTOS Start

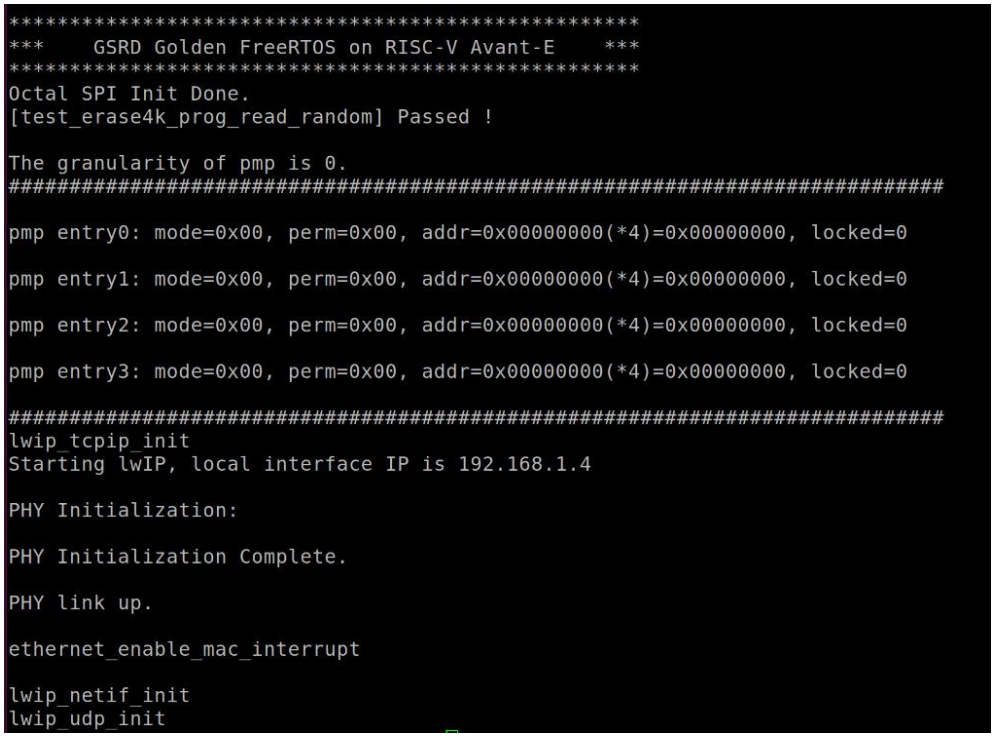


Figure 6.26. Golden GSRD – Output on UART Terminal for FreeRTOS Running

23. Follow the same steps 5 to step 21 to load the Primary GSRD Software and Bitstream. Refer to the Executables section for the folder and file names.
24. The results of running the Primary GSRD are as shown in Figure 6.27 and Figure 6.28.

Figure 6.27. Primary GSRD Bootloader – Output on UART Terminal

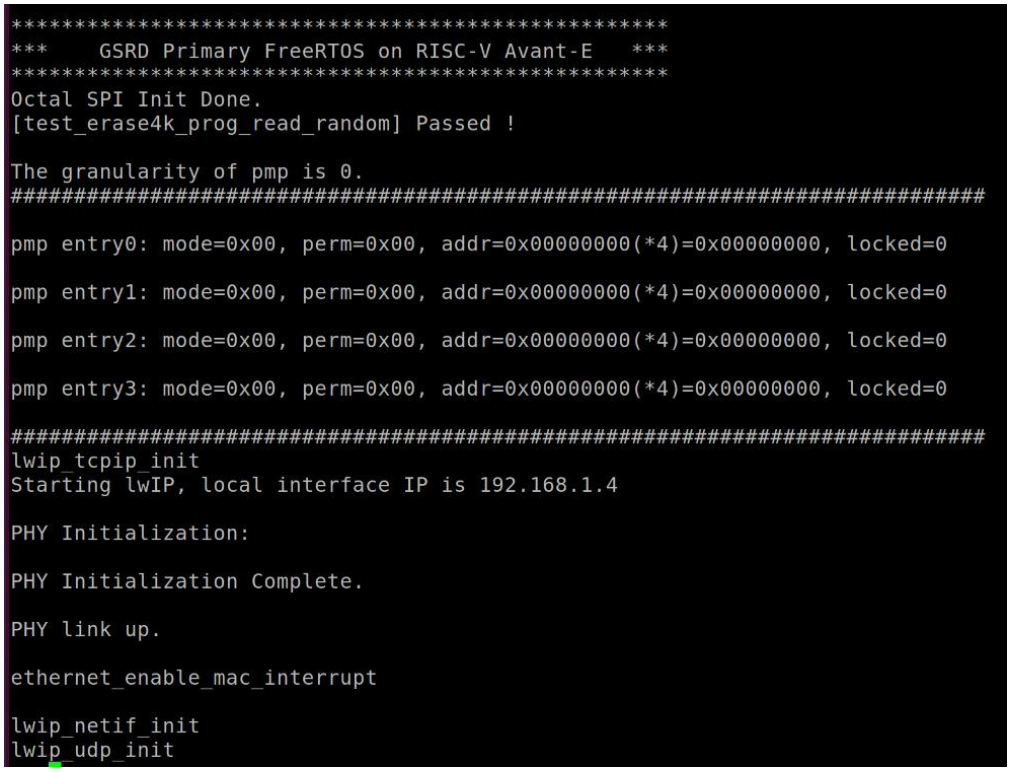


Figure 6.28. Primary GSRD FreeRTOS– Output on UART Terminal

6.6. Programming the Golden, Primary Software, and MCS file

This section demonstrates the multi-boot capability of GSRD by manually booting both Primary and Golden software with CRC checking through programming the MCS file.

To program the golden, primary software, and MCS file, perform the following:

1. Follow Steps 1 to 8 from Programming Standalone Golden or Primary GSRD Bitstream and Application Software section. Do not power-cycle the board.
2. Keep the folder/file of executables handy.
3. To program the **multiboot_system.mcs** file, locate the file in the executables folder that you downloaded and add the file in the Programming file area. The Start Address and End Address are allocated automatically. Confirm the settings as shown in Figure 6.29.

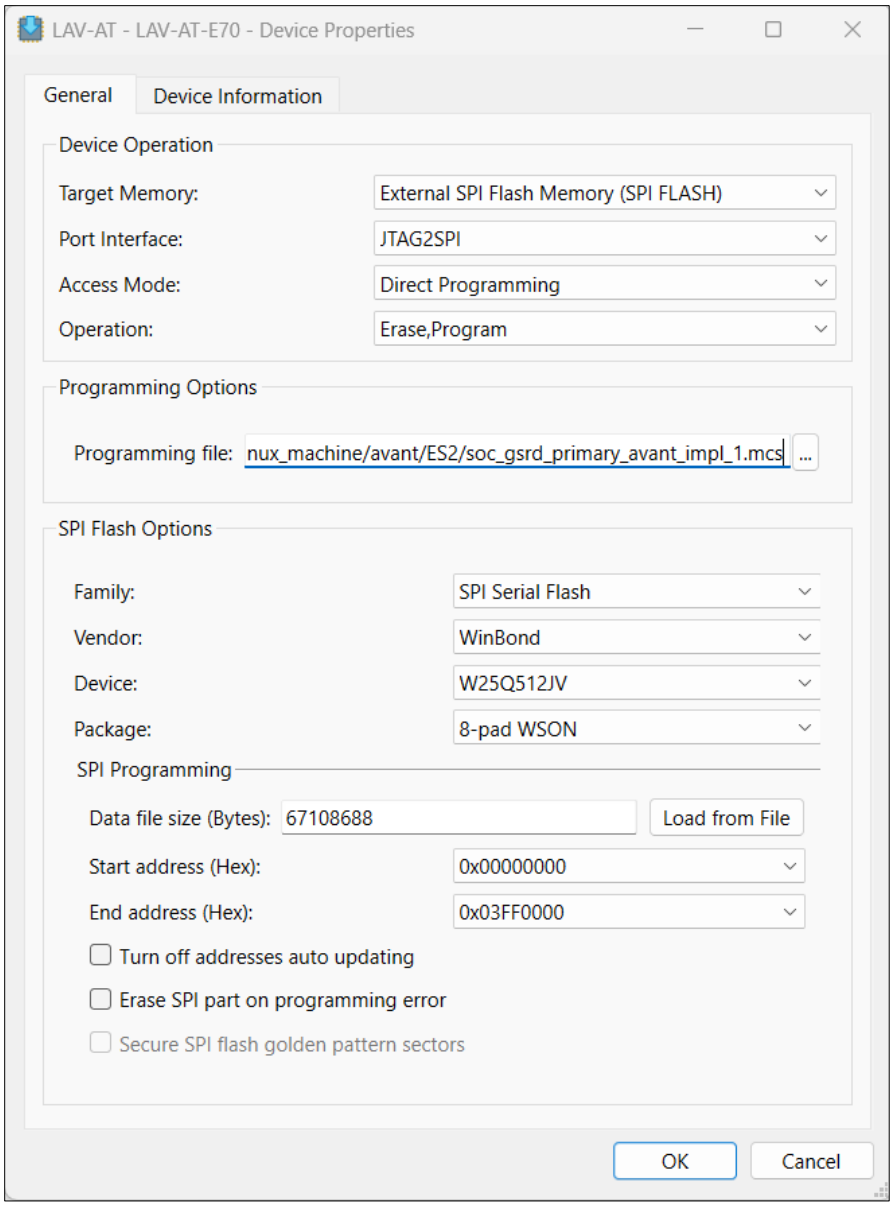


Figure 6.29. Device Properties Window to Setup MCS Programming File

4. Click **OK** and the **Program Device** icon or go to the menu item **Run > Program Device**. Wait until the operation is successful, which takes about 30 to 40 minutes.
5. Do not power-cycle the board yet.
6. Program both the **c_primary_appcrc.bin** and **c_golden_appcrc.bin** files mentioned in the Programming Standalone Golden or Primary GSRD Bitstream and Application Software section. Make sure to confirm that Starting Address (Hex) for both the binaries as per the Executables section.
7. Once both binaries are programmed, turn off power switch **SW6**.
8. Switch **SW7** to **MSPI** mode.
9. Turn on power switch **SW6**.
10. Set up the UART terminal as mentioned in Setting Up the UART Terminal section.
11. Wait for 20 to 30 seconds for the FPGA to load the bitstream from the flash and press **SW1** Reset button.
12. The results in the UART Terminal are displayed as shown in Figure 6.30.

- It loads Primary GSRD software.
- If you press the **SW1** button again, it loads the same Primary GSRD software.

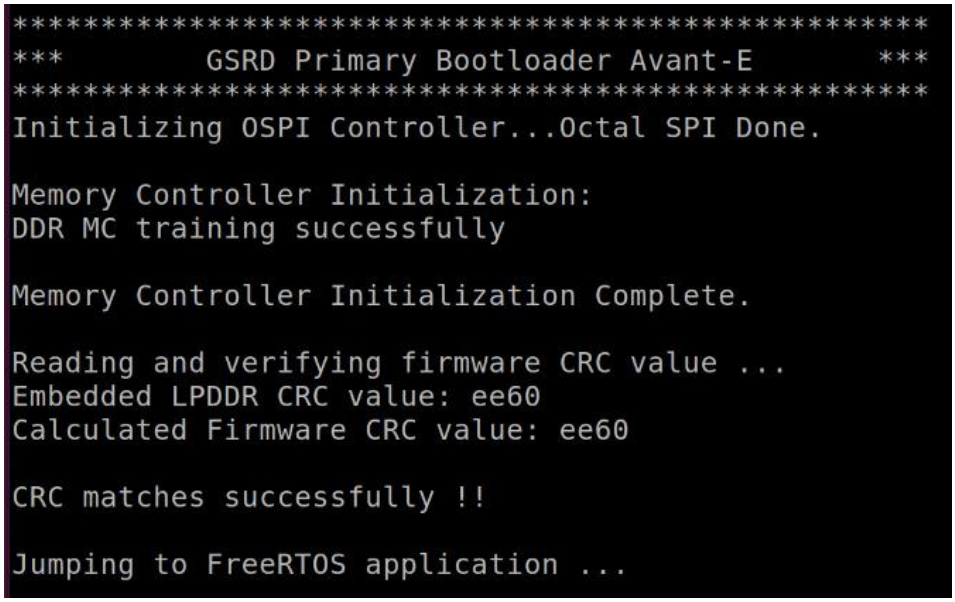


Figure 6.30. UART Terminal Output after Power-Cycling Board with MCS and Binaries Programmed

13. For the manual multi-boot, press **SW2 (PROGRAMN)** button mentioned in the Setting Up the Hardware section on the board just above the **SW1 Reset** button.
14. The results on UART terminal are displayed after a few seconds as shown in Figure 6.31. It switches to the Golden GSRD software and stays there unless the **SW2 PROGRAMN** button is pressed.

Figure 6.31. Switches to Golden GSRD upon SW2 PROGRAMN Button

7. Compiling and Running the Reference Design

This section describes the process of compiling the GSRD/GHRD Reference Design. You can always start with the Primary GHRD/GSRD project, which is a part of the Propel Template. Compilation is required to generate the necessary binary files and bitstreams from the source files. The compilation process involves the following software tools for generating the FPGA bitstream and software executable files. For details, refer to the Lattice Software Tools Requirements section.

These sections show the typical design and compilation flow for GSRD/GHRD design. Hardware validation is performed after the software image is built at each stage.

Table 7.1. List of Actions and Expected Outputs

Actions	Outputs
Building GHRD SoC Project using Lattice Propel SDK and Builder.	sys_env.xml soc_primary_gsrd.sbx soc_golden_gsrd.sbx
Synthesizing the RTL files and generating the bitstream using Lattice Radiant Software	soc_primary_gsrd_impl_1.bit soc_golden_gsrd_impl_1.bit
Building the Hello World Program using Lattice Propel SDK and verifying the RISC-V and System Memory SoC design subsystem is built correctly **Note:** This action is optional	riscv_rtos_helloworld.elf riscv_rtos_helloworld.mem
Building Bootloader binary files using Lattice Propel SDK and verifying the primary or golden bootloader image and SoC design are built correctly	c_primary_bootloader.mem c_primary_bootloader.bin c_golden_bootloader.mem c_golden_bootloader.bin
Building FreeRTOS binary files using Lattice Propel SDK and verifying the primary or golden FreeRTOS image and SoC design are built correctly	c_primary_appcrc.bin c_golden_appcrc.bin
Generating the Multi-Boot MCS File and verifying the multi-boot functionality	multiboot_system.mcs

7.1. Building the GHRD SoC Project Using Lattice Propel SDK and Propel Builder

To build the GHRD SoC design using Propel Template:

1. Create a folder for your project on your PC.
2. Launch Propel SDK application.

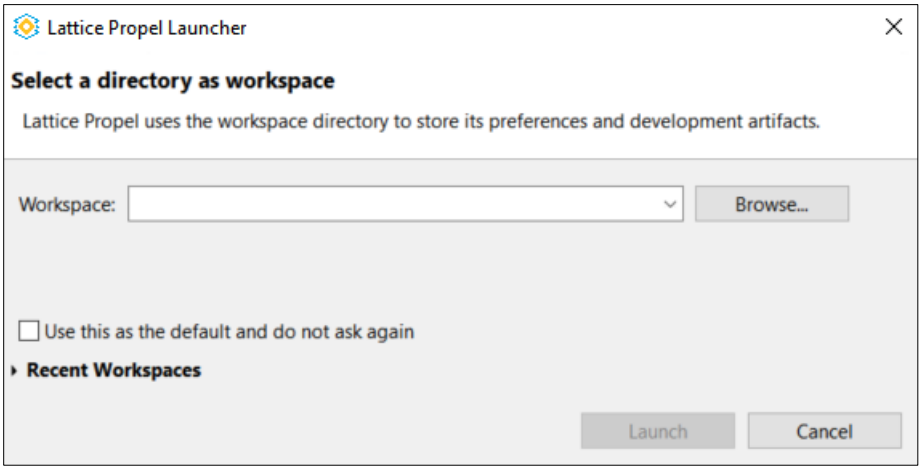


Figure 7.1. Propel SDK Launcher

3. To select the workspace, browse to the created folder by clicking on the **Browse** button as shown in Figure 7.1 and click on **Launch** to create the workspace.

   Note: Name given below is Primary GSRD as you may be using this template to create your own project titles and make modifications on top of Golden SoC/C Templates.

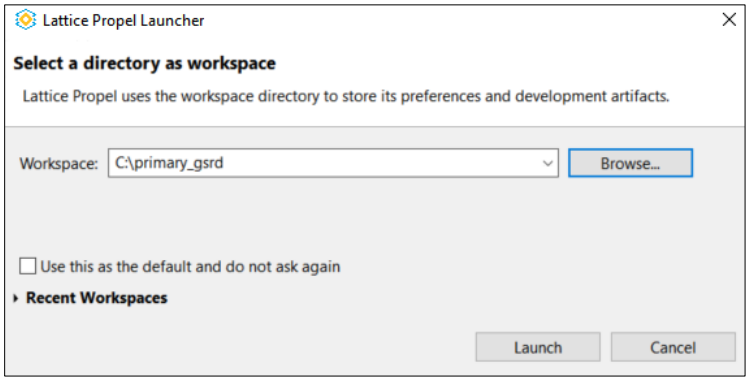


Figure 7.2. Provide Name for the Workspace Directory

4. Click **File > New > Lattice SoC Design Project**.

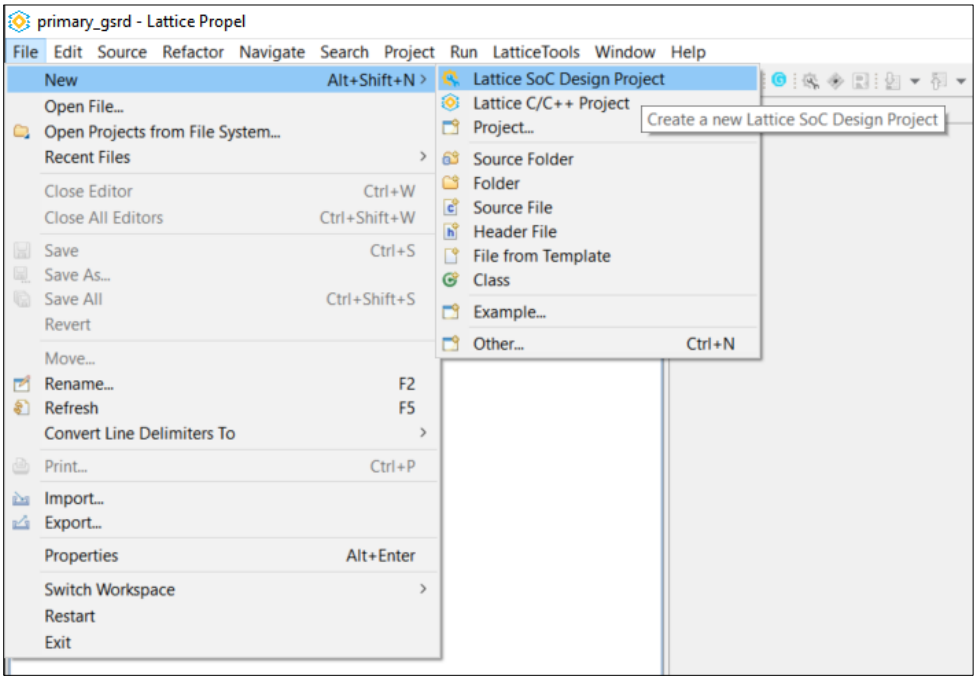


Figure 7.3. Creating Lattice SoC Design Project

5. On the SoC Project window:
   - Enter a name for your SoC project.
   - Enable **Board** option.
   - From **Board Select** section, select **Avant-E Evaluation Board**.
   - From **Processor** section, select **RISC-V RX**.
   - From **Template Design** section, select **GHRD SoC Project LAV-AT** as shown in Figure 7.4.

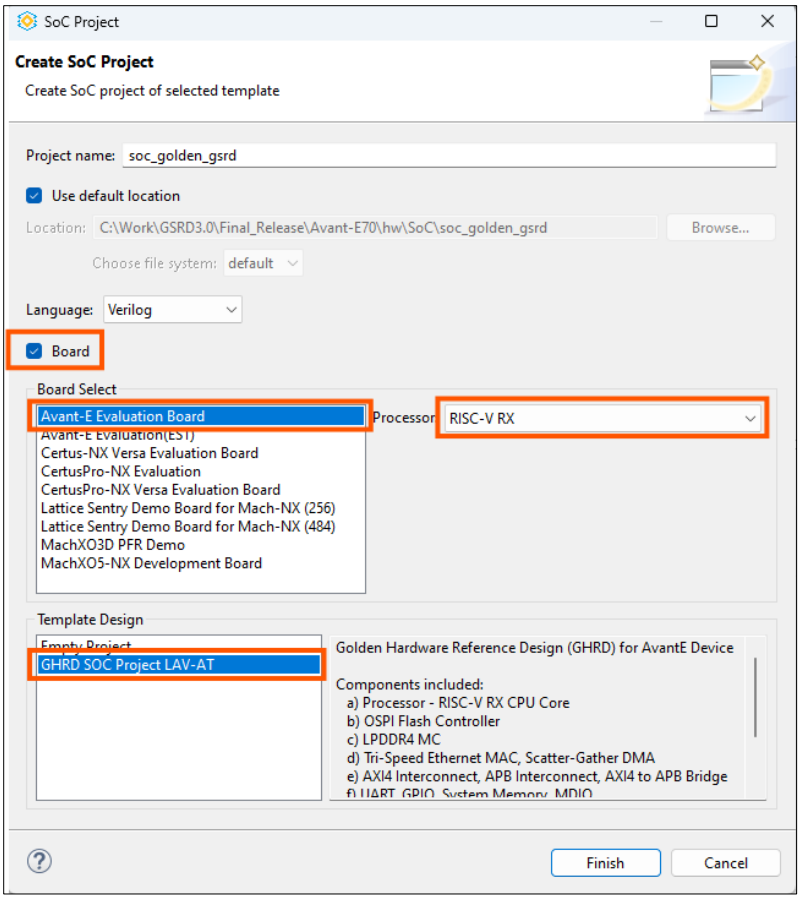


Figure 7.4. SoC Project Window

6. Click **Finish** and it launches the Propel Builder Tool and loads the SoC design based on selected template as above. This generates the HDL files for IPs installed in the project.

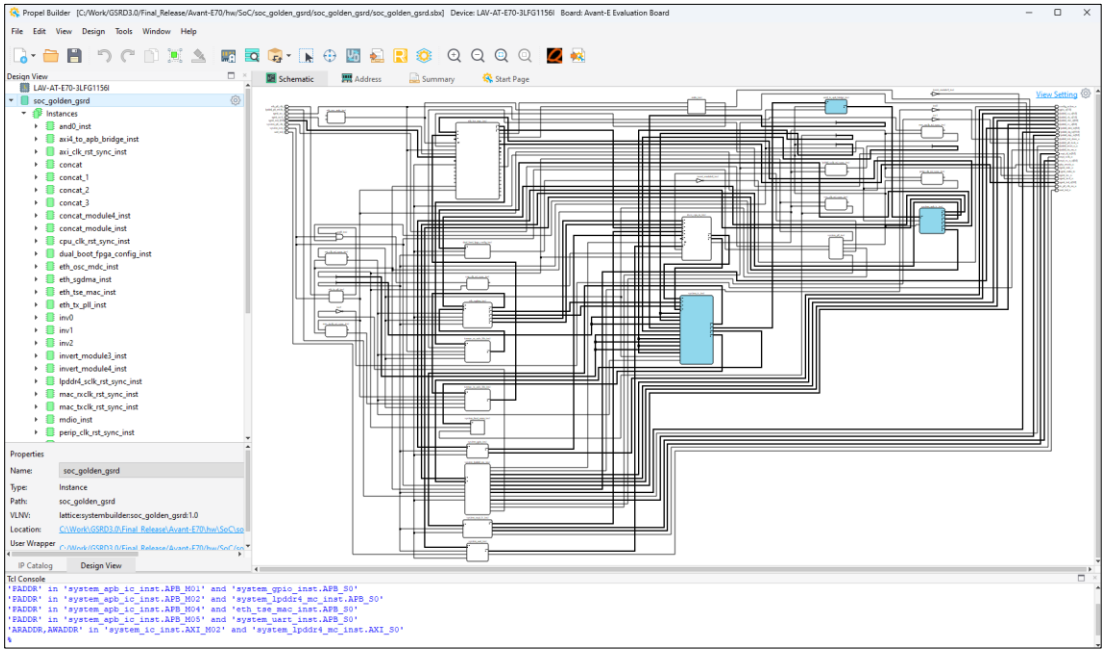


Figure 7.5. Launched SoC in Propel Builder

7. Click on **Design > Validate Design** or Click on below icon as highlighted in screenshot.

Figure 7.6. Validate Design in Propel Builder

The TCL console must be as follows. You should see no error in the **TCL Console** window. The following **INFO** and **WARNINGS** are expected.

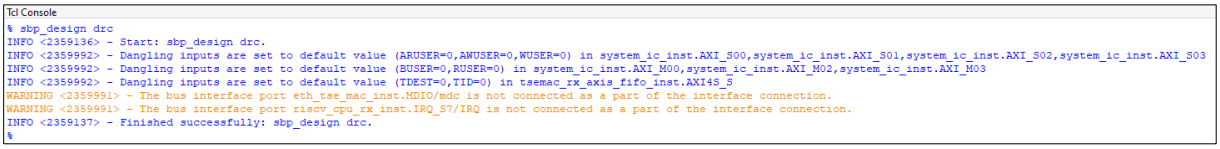


Figure 7.7. TCL Console Output after Validating Design

8. Click on **Design > Generate** or Click on below Icon as highlighted in below screenshot.

Figure 7.8. Generate in Propel Builder

The **TCL Console** must be as follows. Note that the **WARNING** and **INFO** messages are expected.

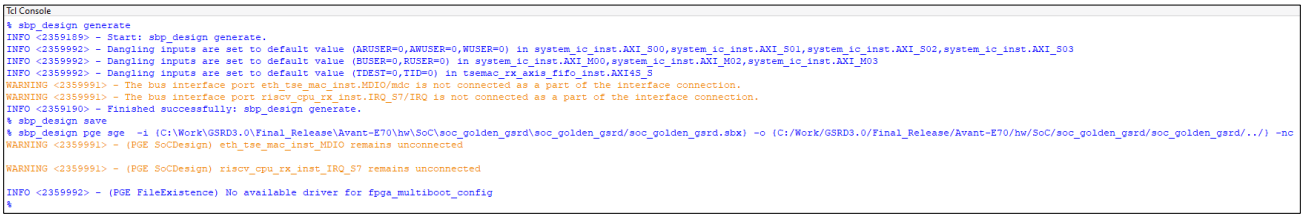


Figure 7.9. TCL Console Output after Generating Design

9. Notice that after Generating the Design, the tool creates the *sys_env.xml* file in the project directory as shown in Figure 7.10. The *sys_env.xml* can be used to generate C/C++ project for Hello World (optional), bootloader and FreeRTOS projects in the next sections.

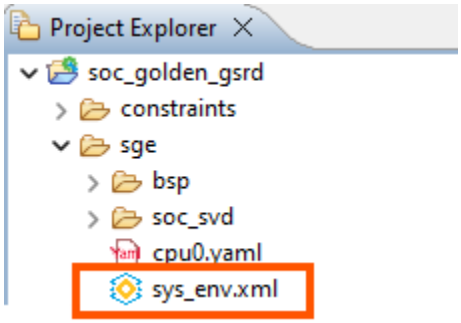


Figure 7.10. sys_env.xml File Created

7.2. Synthesizing the RTL Files and Generating the Bitstream using Lattice Radiant

To synthesize the RTL files and generate the bitstream, perform the following steps:

1. Click the Radiant icon from Propel Builder to launch the Radiant software as shown in Figure 7.11.

Figure 7.11. Run Radiant Icon

2. The Radiant window of your project is launched as shown in Figure 7.12.

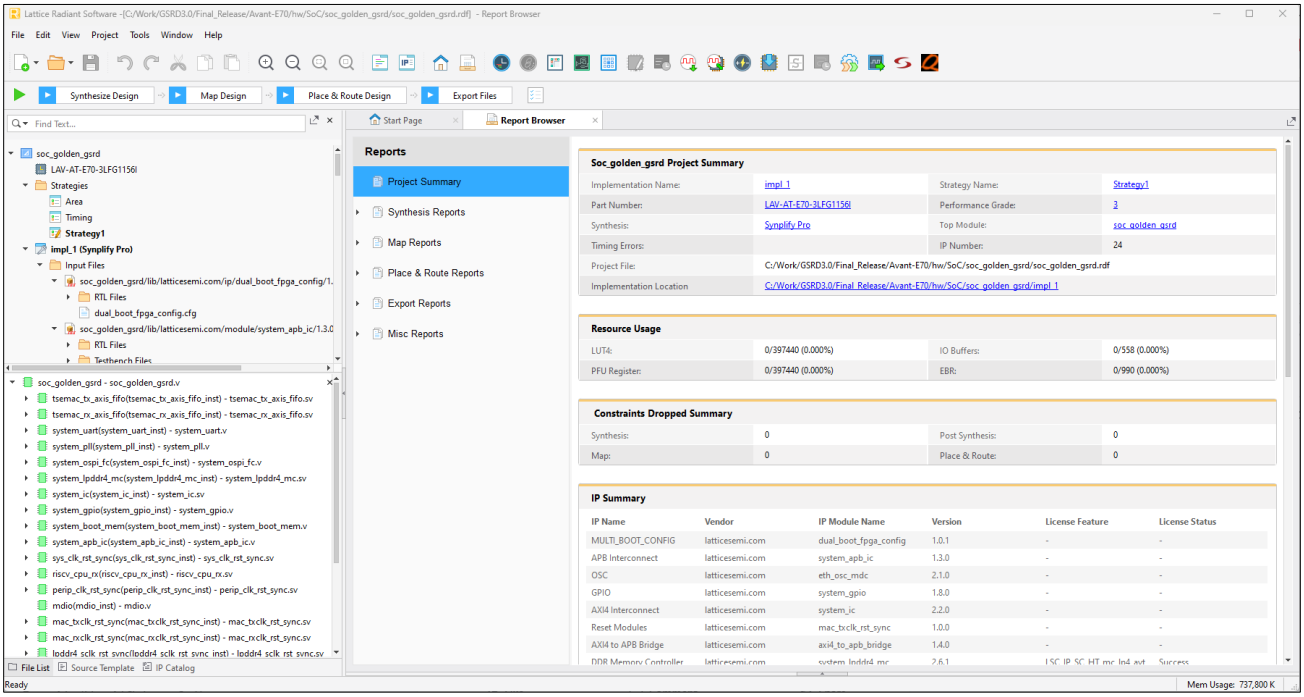


Figure 7.12. Lattice Radiant Window

3. Ensure the *constraints.sdc* and *<project_name>.pdc* are already part of the project. They must respectively appear under pre-synthesis and post-synthesis constraints directory.

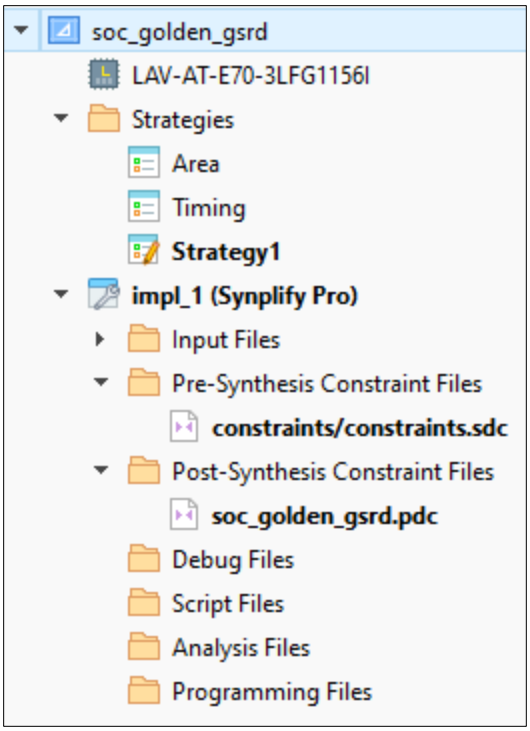


Figure 7.13. Lattice Radiant Window

4. **Place and Route** strategy is used for the GSRD testing.

   Note: You can update these fields as per their machine and run-time requirements. However, due to this change, the Radiant tool might update the warnings and place and route details.

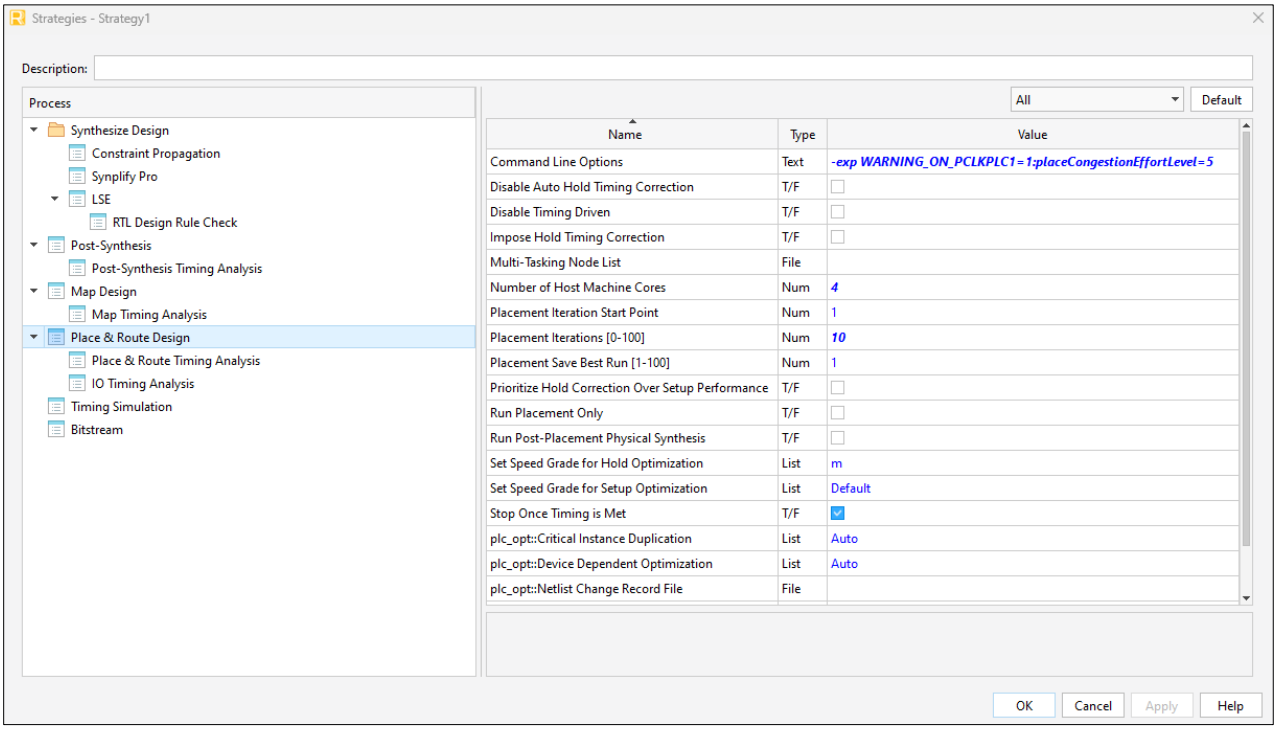


Figure 7.14. Strategy Used for GSRD Testing

5. Click on the **Run All** icon to generate bit file. Wait for the bitstream generation and check the logs.

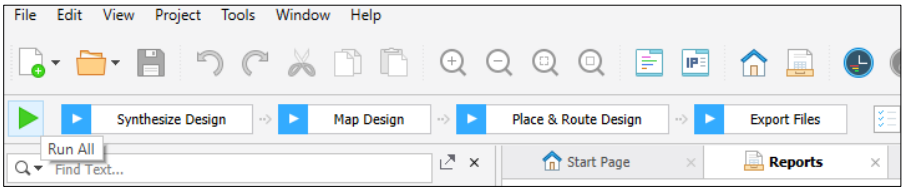


Figure 7.15. Generating the Bit File

6. The compilation flow takes some time to complete. The <project-name_impl_1>.bit file is generated/updated in the project path (*<root_directory>/soc_primary_gsrd/ impl_1*) after compilation is completed successfully as shown in Figure 7.16.

Figure 7.16. Successful Radiant Flow and Bitstream Generation

Note: If you observe timing violations during Lattice Radiant compilation, this may be due to the **Placement Iteration Point number** for **Place and Route** is not working optimally on your machine. In this case, perform the following steps.

   a. In Lattice Radiant software, go to **Project > Active Strategy > Place & Route Design Settings**.
   b. Change **Placement Save Best Run** from 1 to 10. This will force tool to run each iteration and increase runtime.
   c. Unchecked Stop Once Timing is Met option.
   d. Click **OK**.
   e. Click on **Export Files**.

7. These actions cause the **Place and Route Design** with different incremental start point values. The **Place and Route** reports show all ten placement results and select the best timing result.

7.3. Building the Hello World Program Using Lattice Propel SDK (Optional)

For quick start on SoC to boot up, you can create a basic Hello World program by using the template with the Lattice Propel SDK. Note that this action is optional.

1. Create a folder for your project on your PC. For example, HelloWorld.
2. Launch the Propel SDK application.
3. Proceed to the Propel SDK window. Click on **File > New > Lattice C/C++ Project**.

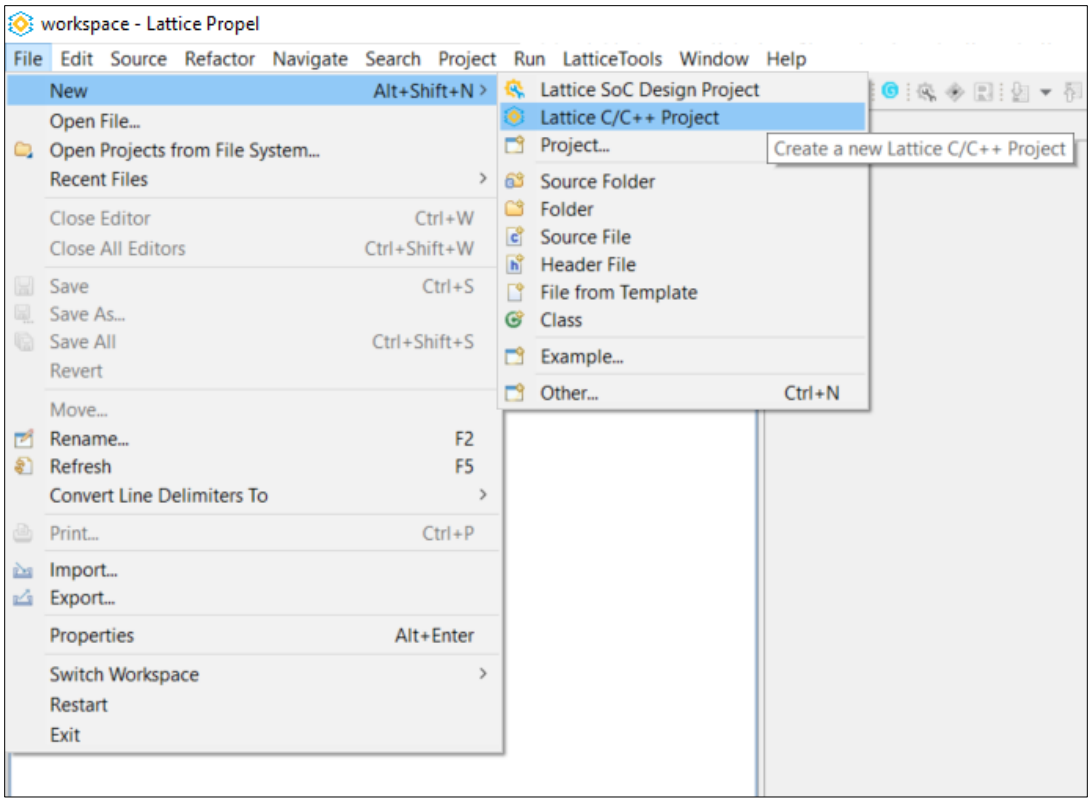


Figure 7.17. Creating Lattice C/C++ Project for Hello World

4. In C/C++ Project window, select the generated *sys_env.xml* file from previous GHRD SoC section. In the *Select Example Application* section, select the **Hello World Project** and provide a name for the project as shown in Figure 7.18.

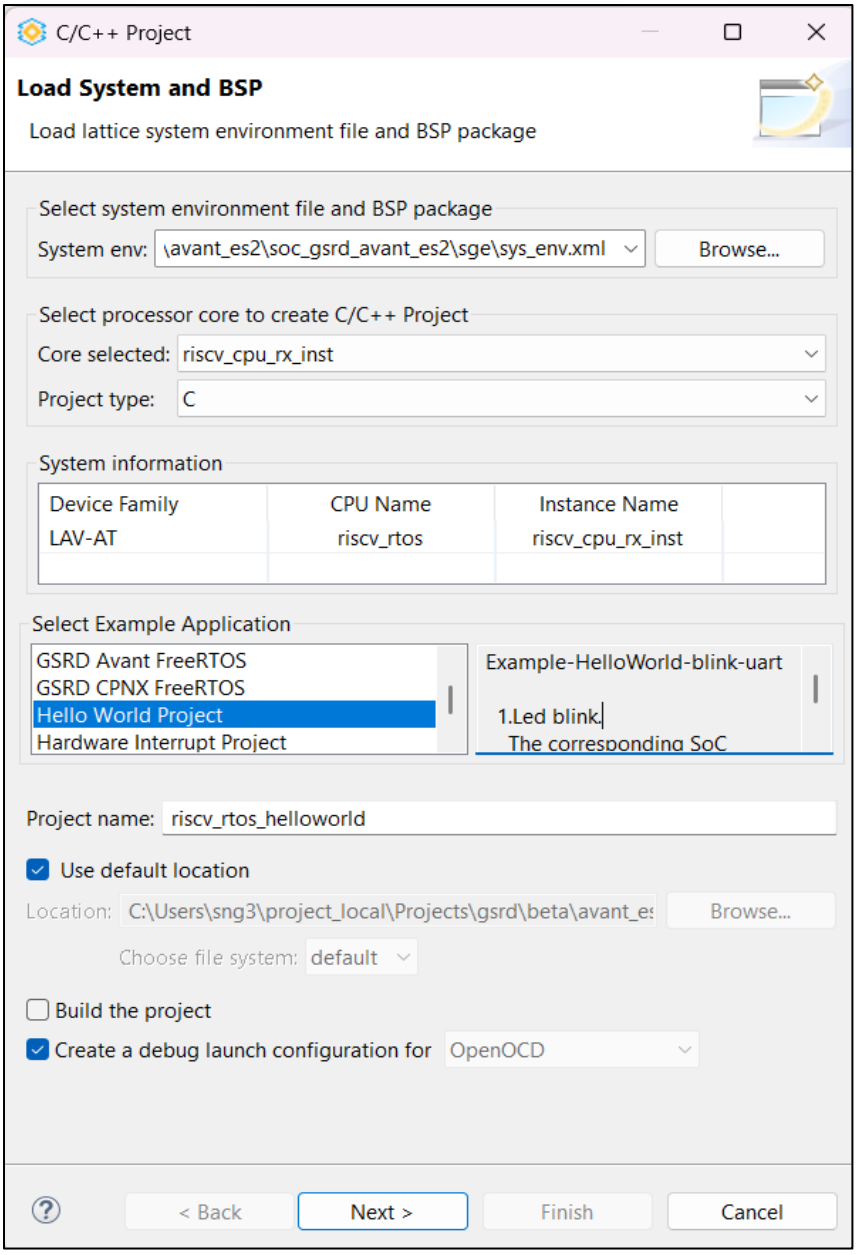


Figure 7.18. Hello World C/C++ Selection

5. Click **Next.** The Lattice Toolchain is shown in Figure 7.19 and click **Finish**.

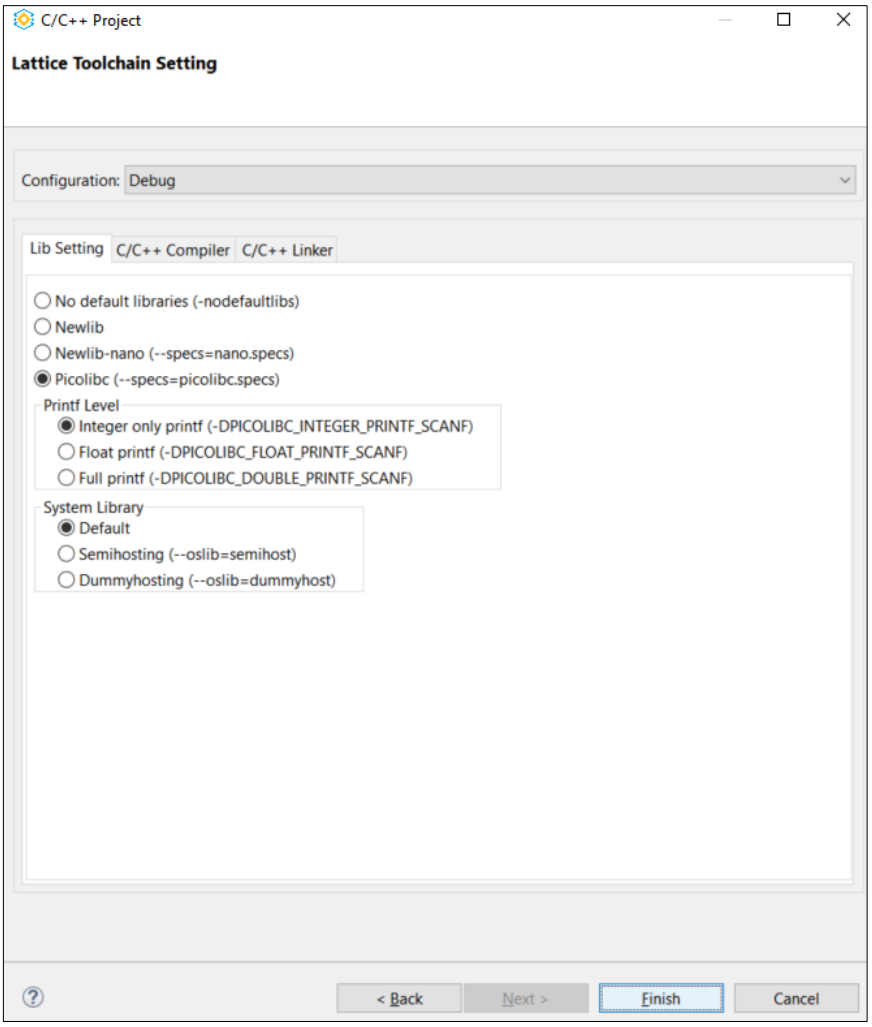


Figure 7.19. C/C++ Lattice Toolchain Setting

6. This loads the Hello World project in the workspace as shown in Figure 7.20.

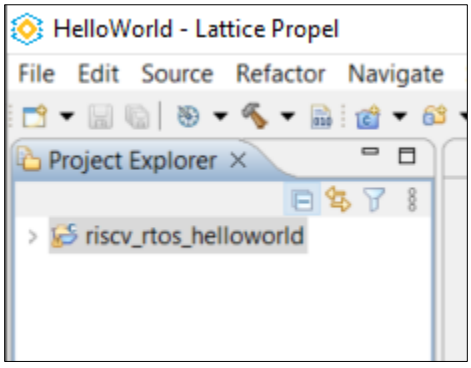


Figure 7.20. Hello World C Project Created

7. To build the Hello World project, right-click on project and select **Build Project**.

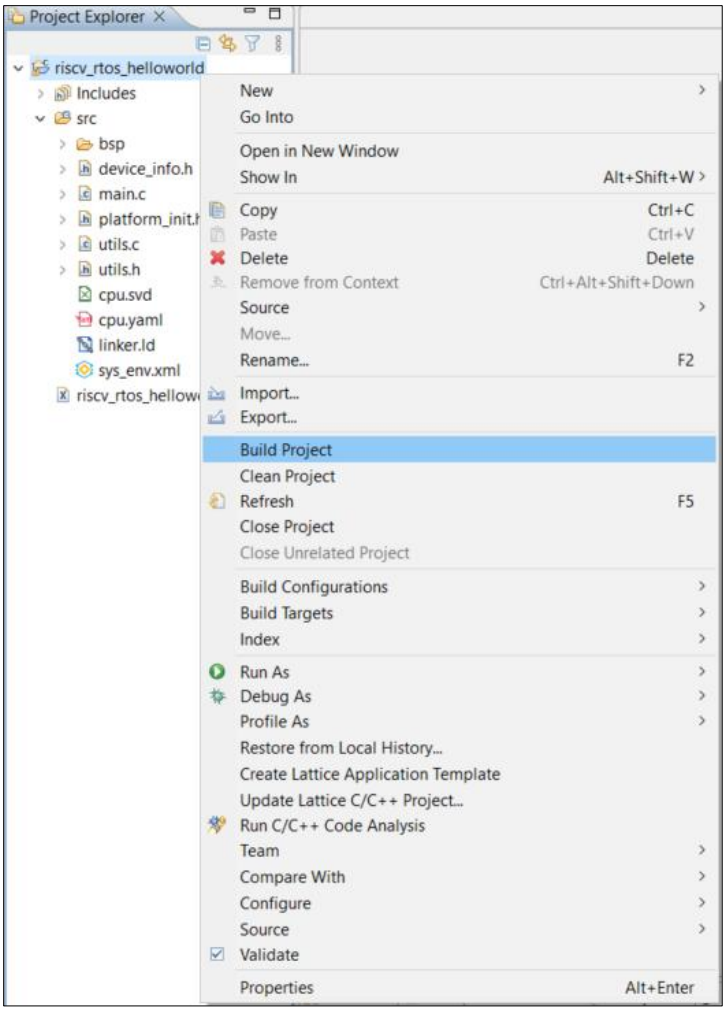


Figure 7.21. Build Hello World Project

8. The console output is displayed as shown in Figure 7.22.

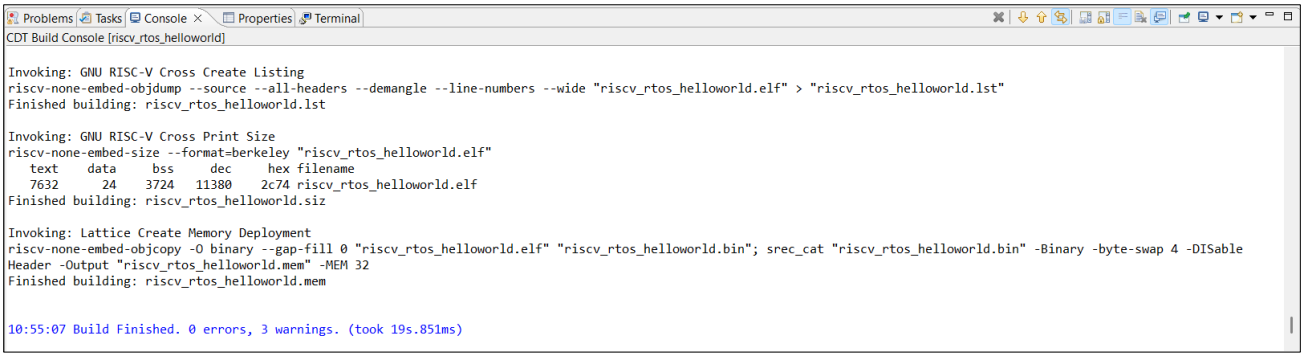


Figure 7.22. Hello World Project Build Console Output

9. The generated *.mem* file can be used to build into the GHRD for quick start-up check on the hardware. Refer to the Using ECO Editor section to integrate the .mem file into the bitstream.
10. Once the bitstream is built with the Hello World program, it can be programmed into the on-board SPI flash for testing. For programming into the flash, refer to the Programming Standalone Golden or Primary GSRD Bitstream and Application Software section.

11. Once programming into the flash is completed, power cycle the board to load the new image that contains the Hello World project.
12. You must see the *Hello World* message as shown in Figure 7.23.

Figure 7.23. Hello World C Program

13. For details on how to create and run the Hello World C program and SoC project, refer to the *Creating a Hello World C Project* section in the Lattice 2025.1 Propel SDK User Guide (FPGA-UG-02234).

7.4. Building the Bare-metal Bootloader Using the Lattice Propel SDK (Primary and Golden)

Once the Hello World program is working, you can proceed to build the Bootloader binary files:

1. Create a folder for your project on your PC. Launch the Propel SDK application.
2. Proceed to the Propel SDK window. Click on **File > New > Lattice C/C++ Project**.

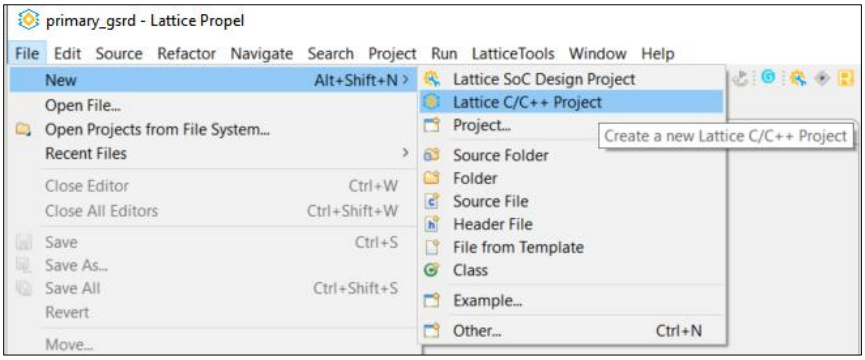


Figure 7.24. Creating Lattice C/C++ Project for Bootloader

3. In C/C++ Project window, it automatically selects the generated *sys_env.xml* file. In the *Select Example Application* section, select the **GSRD Avant Bootloader** and provide a name for the bootloader as shown in Figure 7.25.

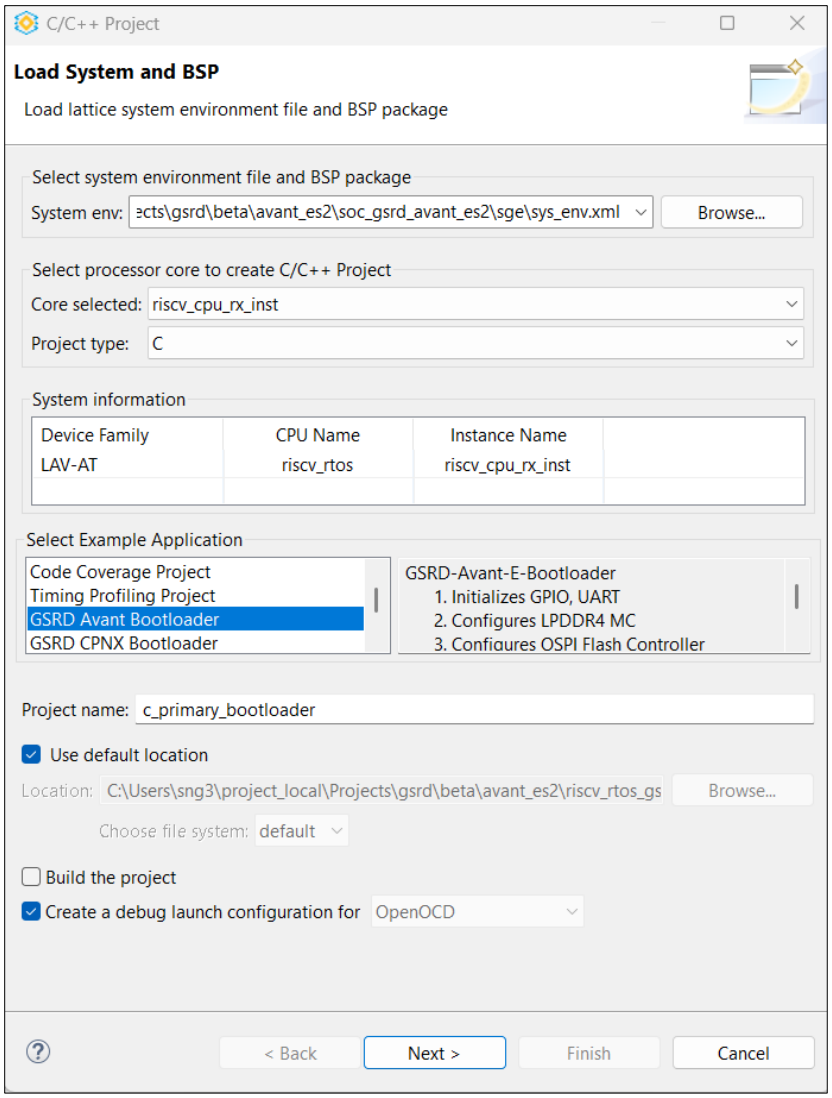


Figure 7.25. Bootloader C/C++ Selection

4. Click **Next.** The Lattice Toolchain is shown in Figure 7.26 and click **Finish**.

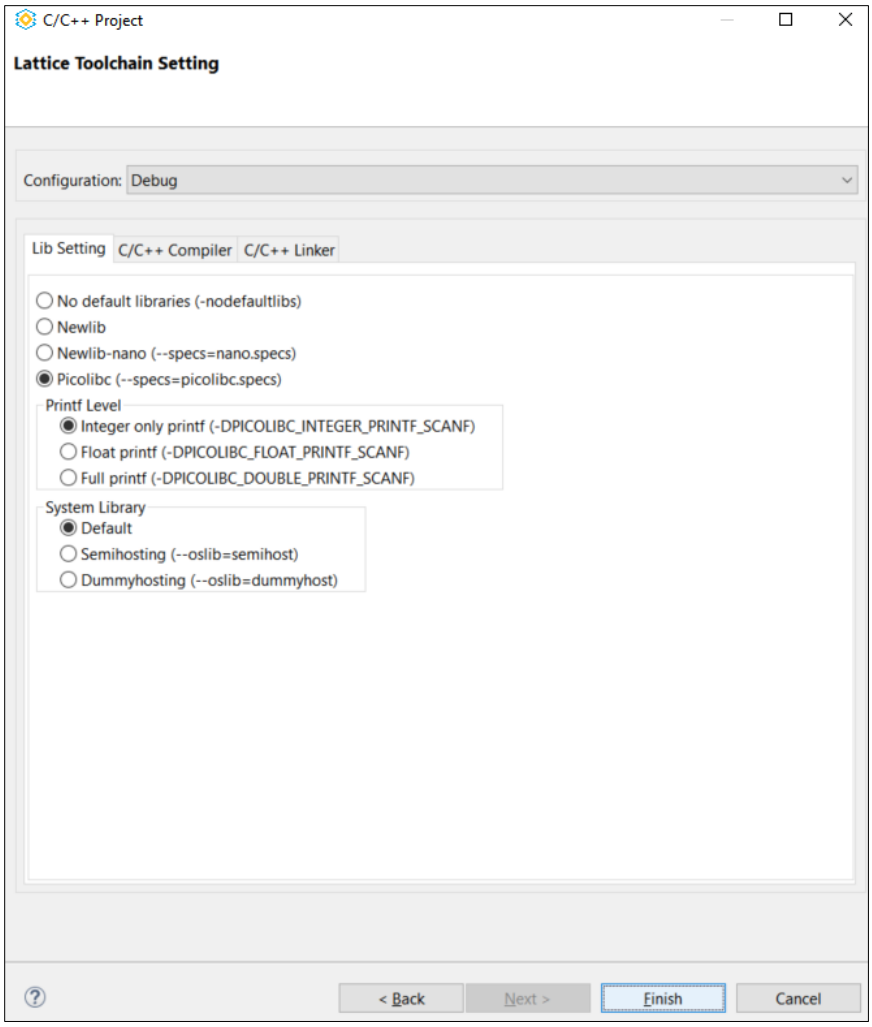


Figure 7.26. C/C++ Lattice Toolchain Setting

5. This loads the bootloader project in the workspace as shown in Figure 7.27.

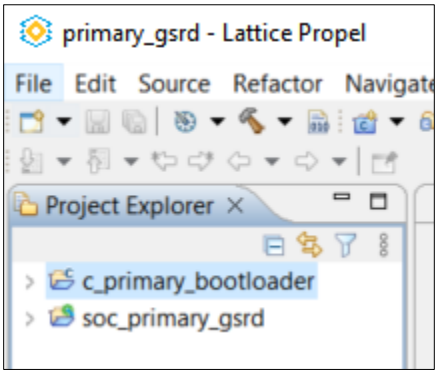


Figure 7.27. Bootloader C Project Created

6. To build for the Primary Bootloader, you need to define _PRIMARY_BUILD_ in the start of the main.c file. This enables the SPI address to point to Primary FreeRTOS application software image in the SPI flash. Refer to Appendix C. Using Different SPI Flash Manufacturer in GSRD Bare-metal Bootloader to modify the Octal SPI Controller driver according to the flash device used.

```
54 /* Set for GOLDEN OR PRIMARY build */
55 #define _PRIMARY_BUILD_
56
```

Figure 7.28. Primary Build Define – Set _PRIMARY_BUILD_ for Primary Build

7. To create the bootloader binary file (**c_primary_bootloader.mem)**, right-click on **c_primary_bootloader** and select **Build Project**.

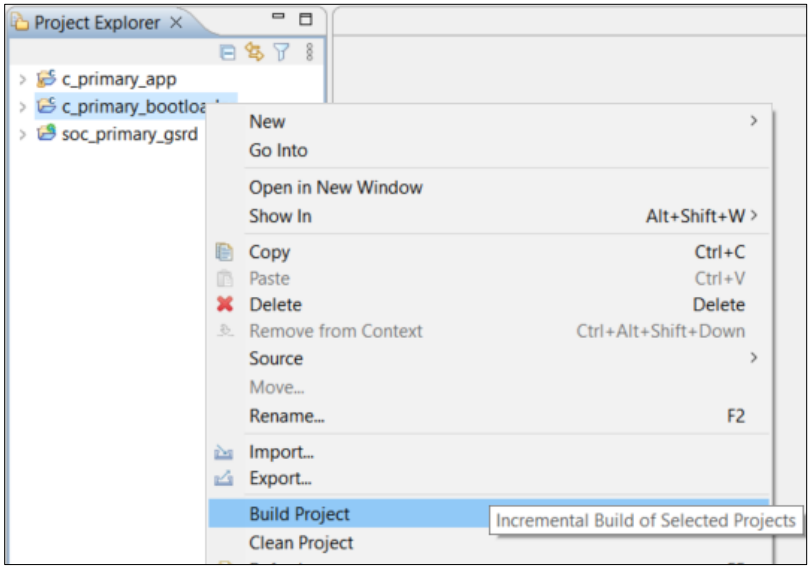


Figure 7.29. Build Bootloader Project

8. The console output is displayed as shown in Figure 7.30.

```
Invoking: GNU RISC-V Cross Create Listing
riscv-none-embed-objdump --source --all-headers --demangle --line-numbers --wide "c_primary_bootloader.elf" > "c_primary_bootloader.lst"
Invoking: GNU RISC-V Cross Print Size
Invoking: Lattice Create Memory Deployment
riscv-none-embed-size --format=berkeley "c_primary_bootloader.elf"
riscv-none-embed-objcopy -O binary --gap-fill 0 "c_primary_bootloader.elf" "c_primary_bootloader.bin"; srec_cat "c_primary_bootloader.bin" -Binary -byte-swap 4 -DISable Header -Output
"c_primary_bootloader.mem" -MEM 32
   text    data     bss     dec     hex filename
  26696      32   11840   38568    96a8 c_primary_bootloader.elf
Finished building: c_primary_bootloader.siz

Finished building: c_primary_bootloader.lst

Finished building: c_primary_bootloader.mem


15:17:00 Build Finished. 0 errors, 3 warnings. (took 17s.676ms)
```

Figure 7.30. Bootloader Build Project Console Output

9. This creates a debug folder as shown in Figure 7.31. The binary created is named c_primary_bootloader.mem. This goes as a part of System Memory in Propel Builder SoC design. This can be done through ECO editor or directly from the Propel Builder System Memory user interface.

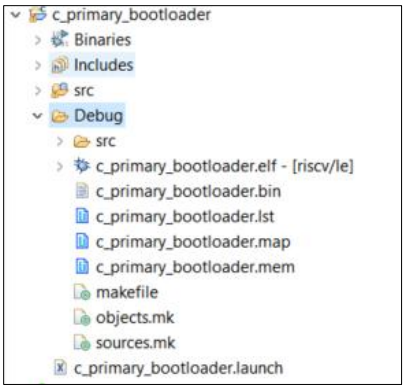


Figure 7.31. Bootloader Binary Created

10. Update the system memory content by using ECO editor. Refer to the Using ECO Editor section for more information.
11. Once the primary bootloader is run, user can see the following output. The output message is expected, and users can proceed to build FreeRTOS application in the next section.

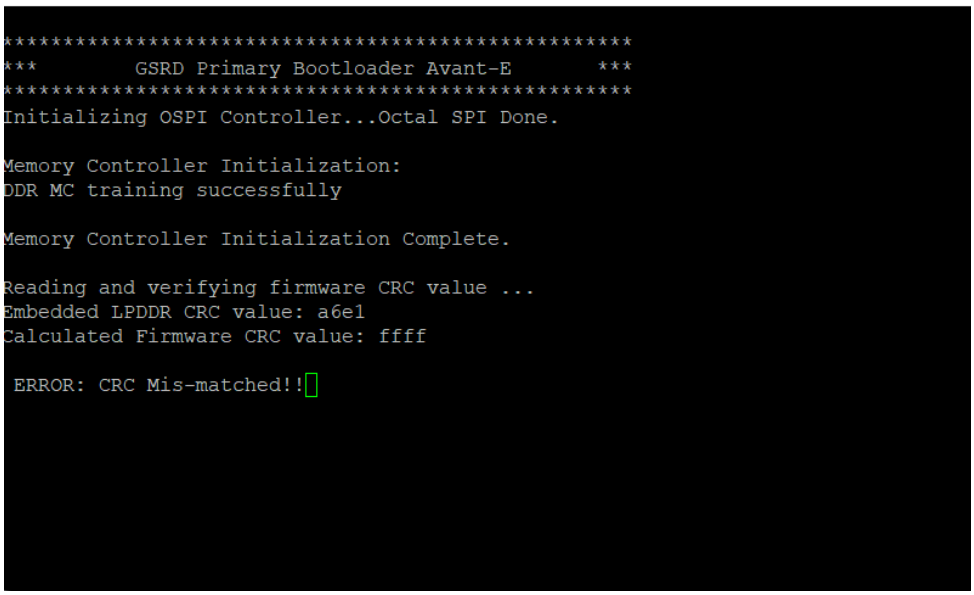


Figure 7.32. Bootloader boots up without FreeRTOS application

12. In the case to create the golden bootloader **c_golden_bootloader.mem,** set the **#define _GOLDEN_BUILD_** in the *main.c* file to build for golden bootloader binary file.

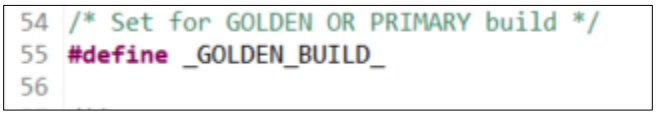


Figure 7.33. Golden Build Define – Set _GOLDEN_BUILD_ for Golden Build

13. Repeat steps 1 to 11 to create the golden bootloader project.

7.5. Building the FreeRTOS Application Software using Lattice Propel SDK (Primary and Golden)

To build the FreeRTOS application software using Lattice Propel SDK, perform the following:

1. Similarly, create the FreeRTOS C Project. Go to **File > New > Lattice C/C++ Project**.

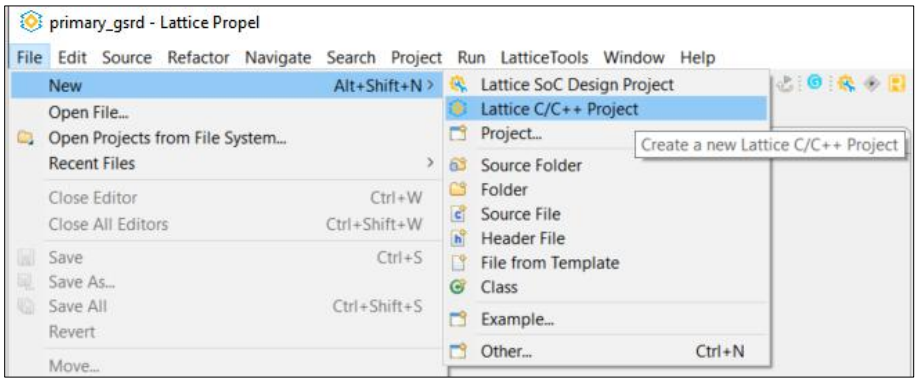


Figure 7.34. Creating C/C++ Project for FreeRTOS

2. In the *Select Example Application*, select the **GSRD Avant FreeRTOS** project. Provide the project name as shown in Figure 7.35.

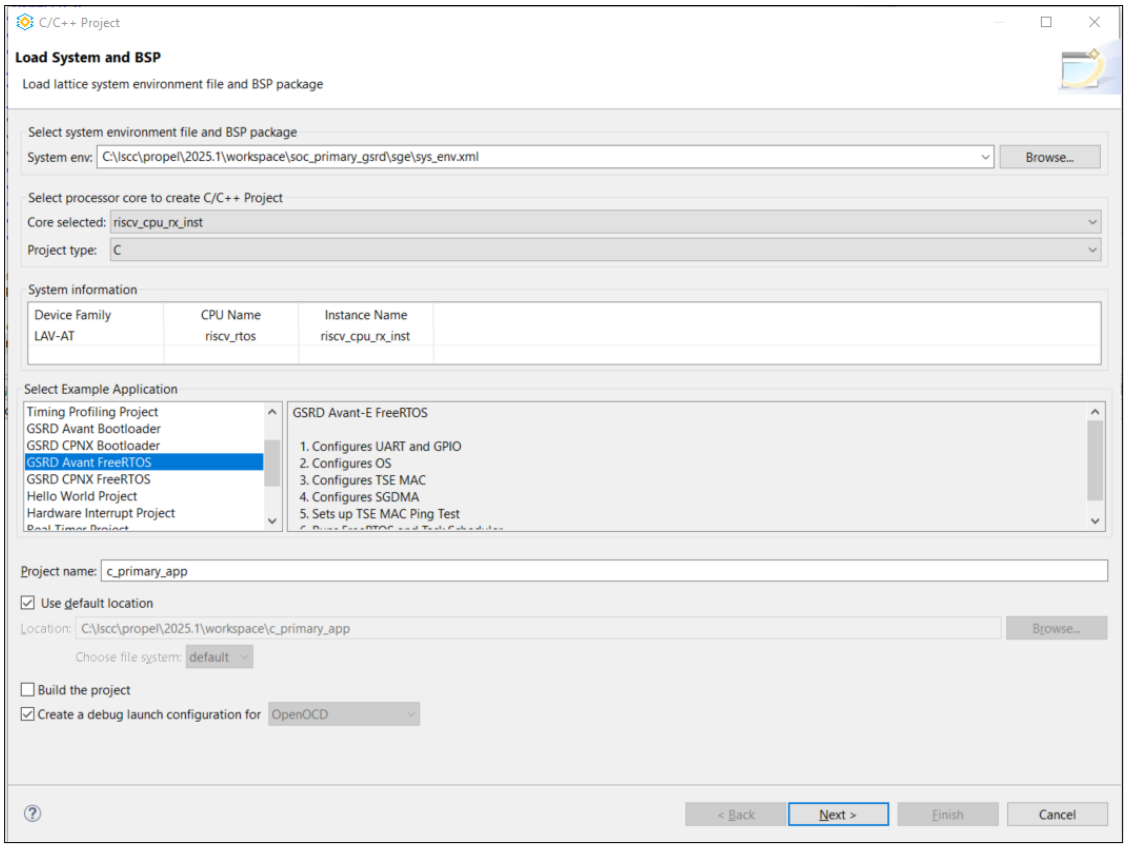


Figure 7.35. FreeRTOS C/C++ Selection

3. Click **Next** and **Finish**.

   Note: This is a manual step you need to perform. After the application project is created, it includes the *crc_add_debug.txt* and *crc_add_release.txt* as shown in Figure 7.36.

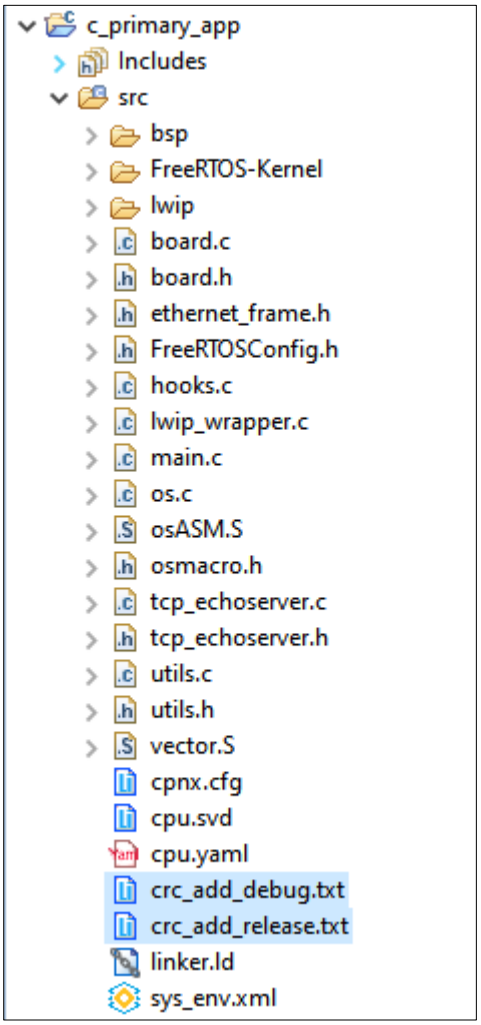


Figure 7.36. FreeRTOS Project Created

4. To avoid any confusion, move the following two files from c_primary_app > src to *c_primary_app* directory.

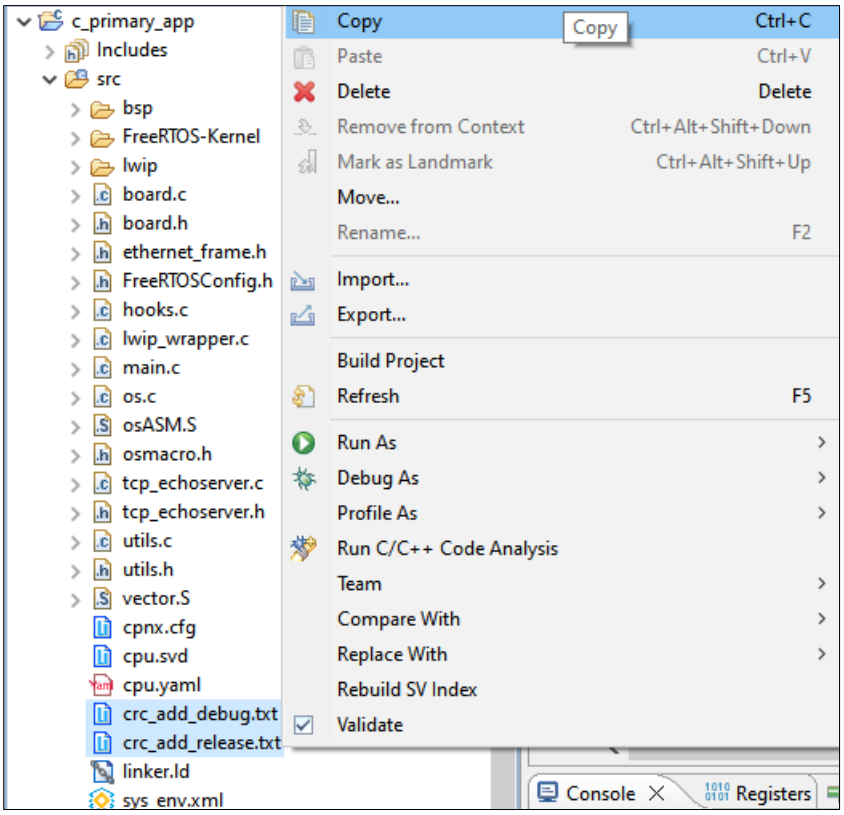


Figure 7.37. Copy the CRC Add files

5. Paste it in your main c_primary_app project as shown in Figure 7.38.

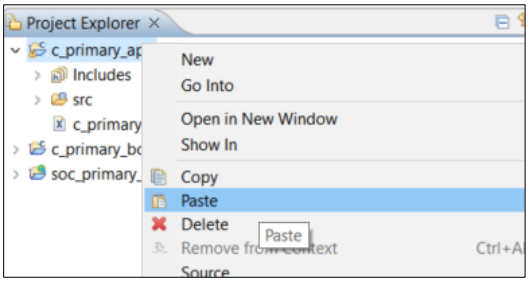


Figure 7.38. Paste in FreeRTOS C project

6. The text files are now added under the FreeRTOS App C project as shown in Figure 7.39.

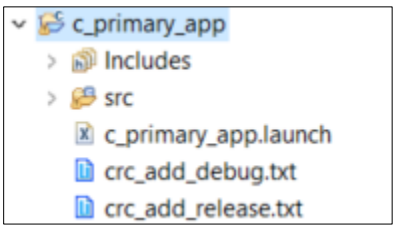


Figure 7.39. Copied Text Files

7. To build the Primary GSRD, you need to define _PRIMARY_BUILD_ in the start of the *main.c* file. This enables the SPI address to point to Primary FreeRTOS image in the SPI flash.

```
54 /* Set for GOLDEN OR PRIMARY build */
55 #define _PRIMARY_BUILD_
56
```

Figure 7.40. Primary Build Define – Set _PRIMARY_BUILD_ for Primary Build in main.c

8. In the c_primary_app folder, update the *crc_add_debug.txt* or *crc_add_release.txt* file as shown in Figure 7.41. Line 5 and line 12 must be replaced with the app project name that you provided; in this case the name is c_primary_app. Hence, the name of the file is replaced with *c_primary_app.bin*. Line 16 must contain the name of c_primary_appcrc.bin. This output file has the CRC for application software appended at the end of binary file.

crc_add_debug.txt

```
1 # srec_cat command file to add the CRC and produce application file to be flashed
2 # Usage: srec_cat @filename
3
4 #first: create CRC checksum
5 ..\Debug\c_primary_app.bin -Binary
6 -fill 0xFF 0x0000 0x40000          # fill code area with 0xff
7 -crop 0x0000 0x3fffe               # just keep code area for CRC calculation below
8 -CRC16_Big_Endian 0x3fffe -CCITT   # calculate big endian CCITT CRC16 at given address.
9 -crop 0x3fffe 0x40000              # keep the CRC itself
10
11 #second: add application file
12 ..\Debug\c_primary_app.bin -Binary
13 -fill 0xFF 0x0000 0x3fffe          # fill code area with 0xff
14
15 # generate a Binary file
16 -o ..\Debug\c_primary_appcrc.bin -Binary
```

Figure 7.41. Update crc_add_debug.txt

9. Open the *Linker.ld* file from *c_primary_app*.

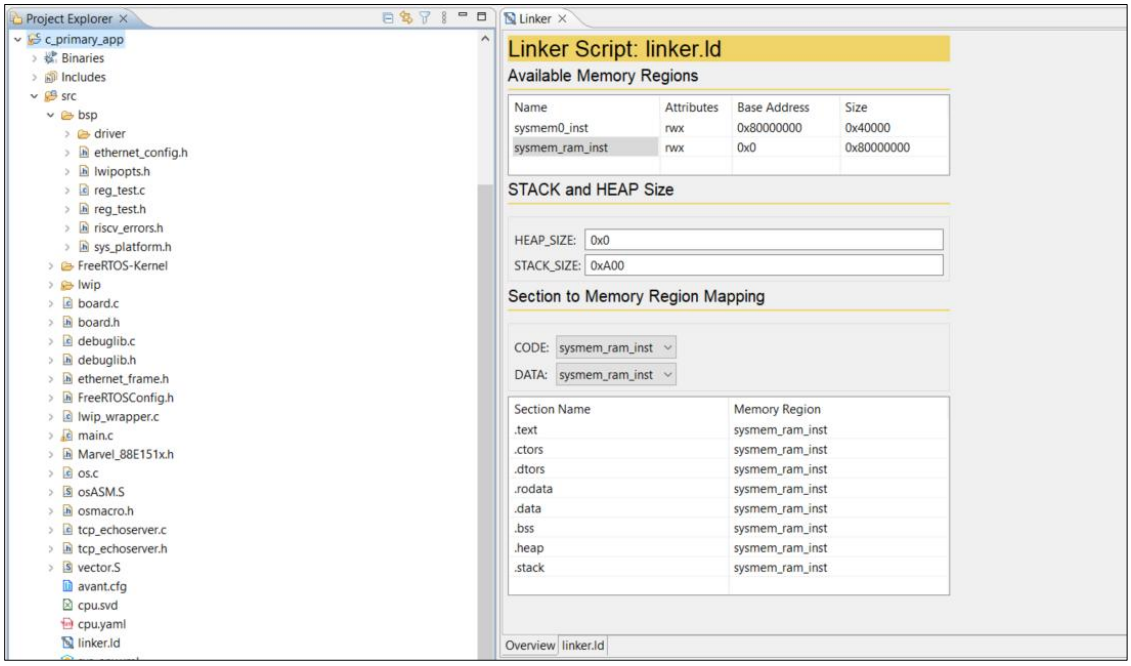


Figure 7.42. Open linker.ld File

10. Confirm the MEMORY org address is set to **0x0000000** for FreeRTOS to run from LPDDR4 memory.

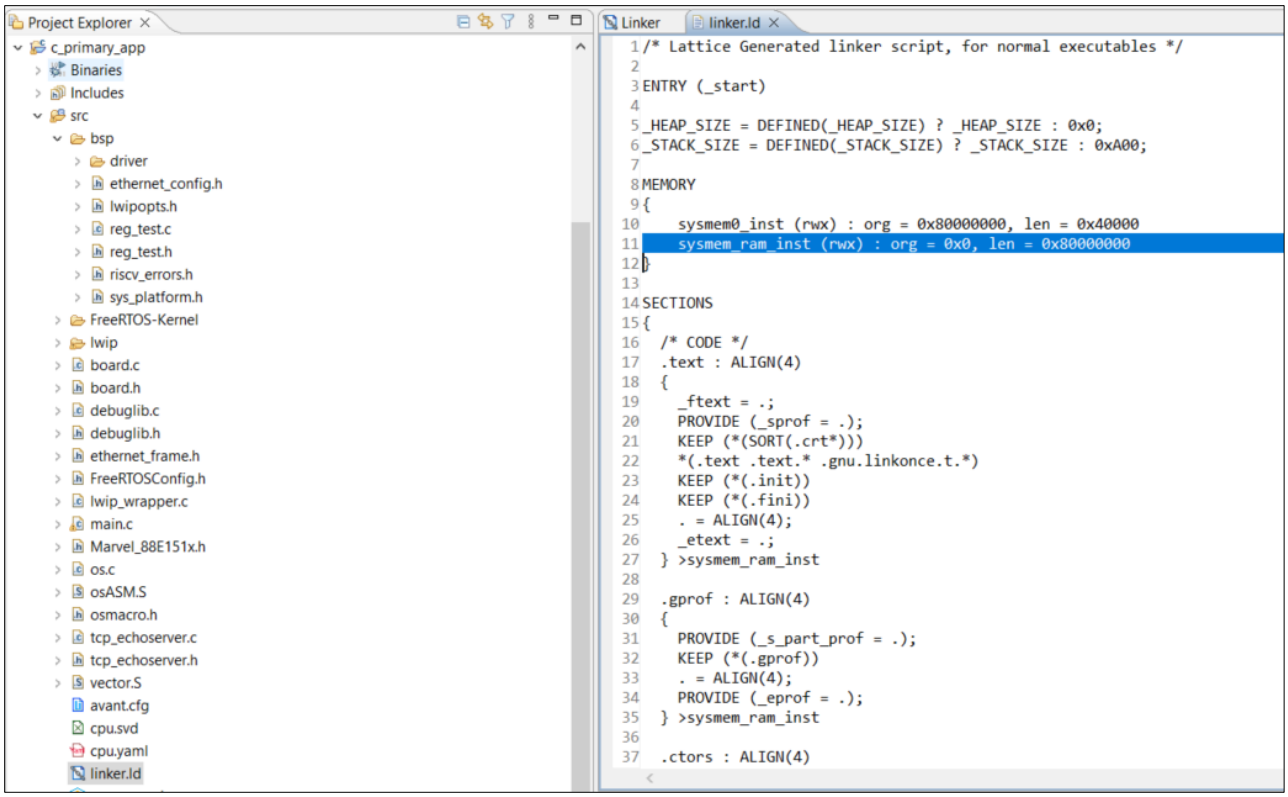


Figure 7.43. Update linker.ld File

11. Press **Ctrl + s** to save the change as shown in Figure 7.44.

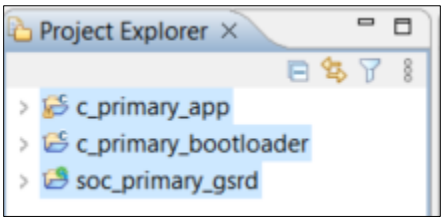


Figure 7.44. Workspace

12. Before creating the binary for FreeRTOS application project, right-click on **c_primary_app** project and click **Properties**.

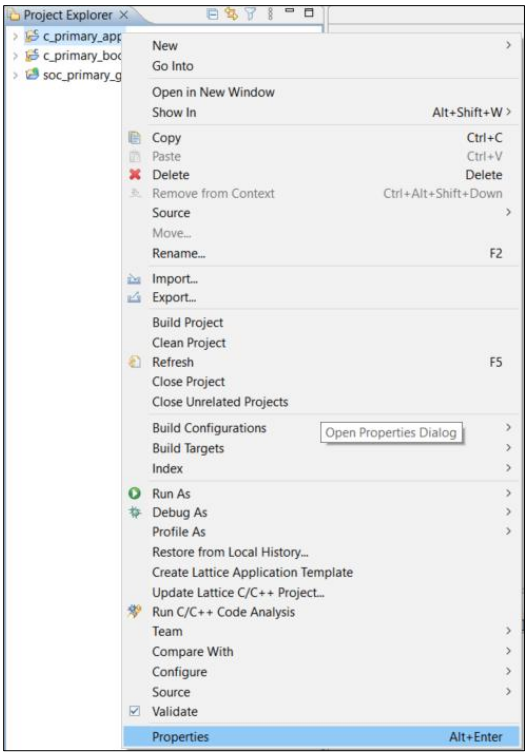


Figure 7.45. Properties

13. In case you wish to create a release folder on the C project build, go to **C/C++ Build > Settings**.
14. Click on **Manage Configurations** and select **Release**. Click **Set Active**, then click **OK**.

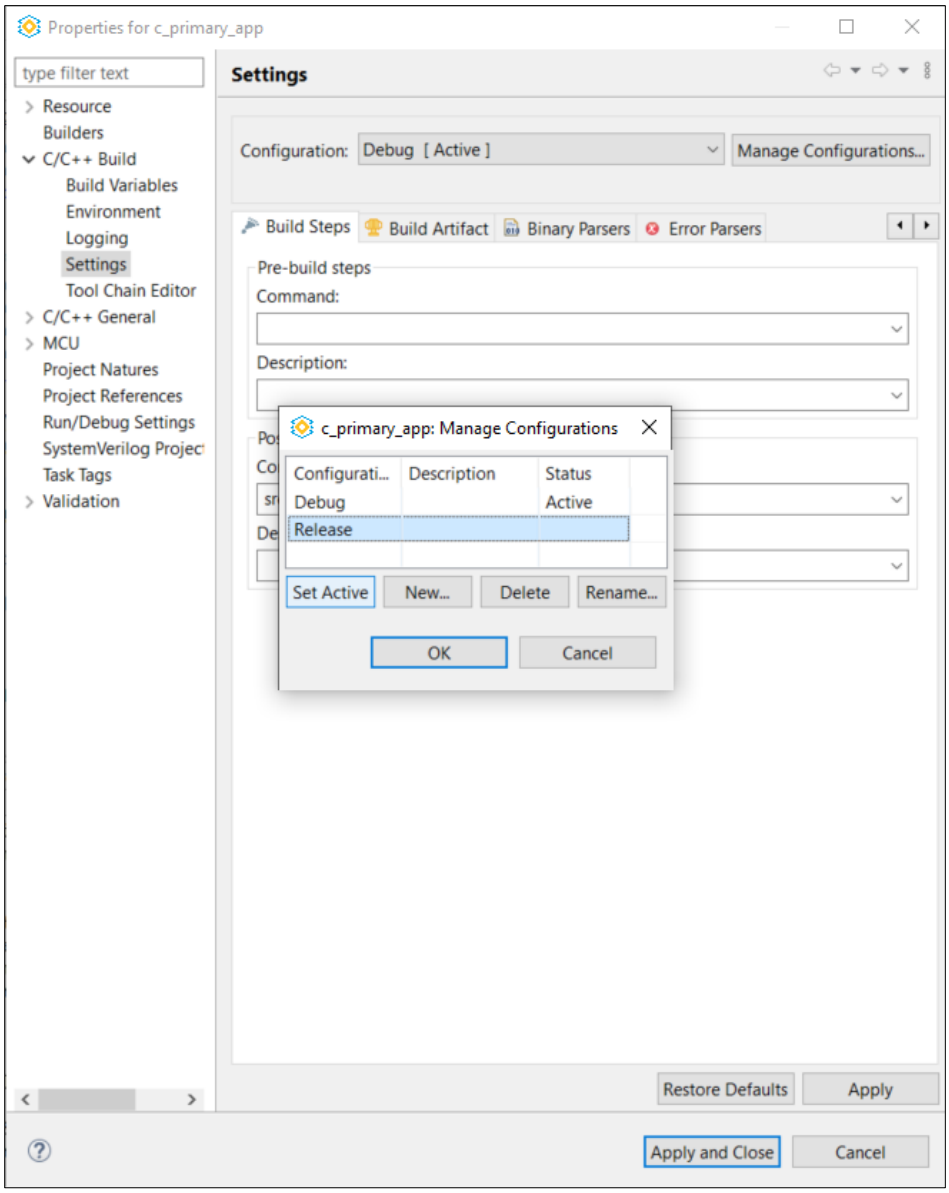


Figure 7.46. Set Release as Active Configuration

15. Go to **C/C++ Build > Settings > Build Steps** and under **Post-Build Steps > Command**, add these commands as shown below for your respective builds, that is for **Debug** or **Release**.

For Windows system:

```
srec_cat.exe "@..\crc_add_debug.txt"

srec_cat.exe "@..\crc_add_release.txt"
```

For Linux system:

```
srec_cat "@../crc_add_debug.txt"

srec_cat "@../crc_add_release.txt"
```

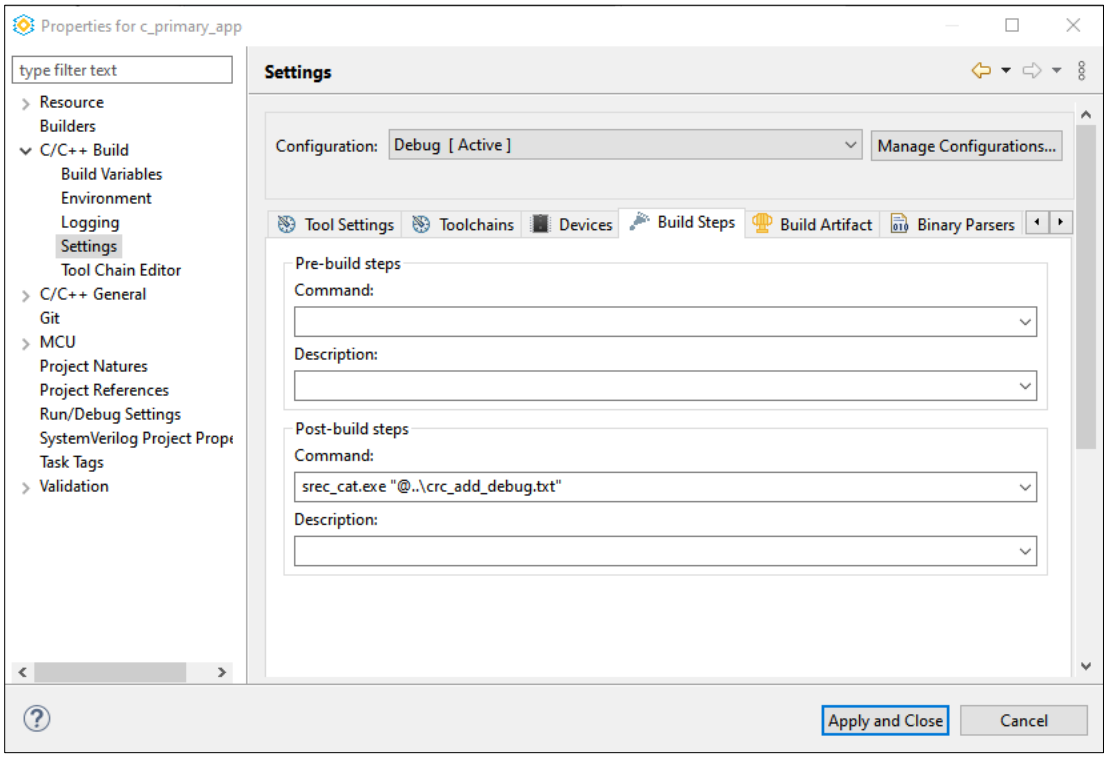


Figure 7.47. Adding Post Build Step for FreeRTOS Application CRC Binary Append (Windows)

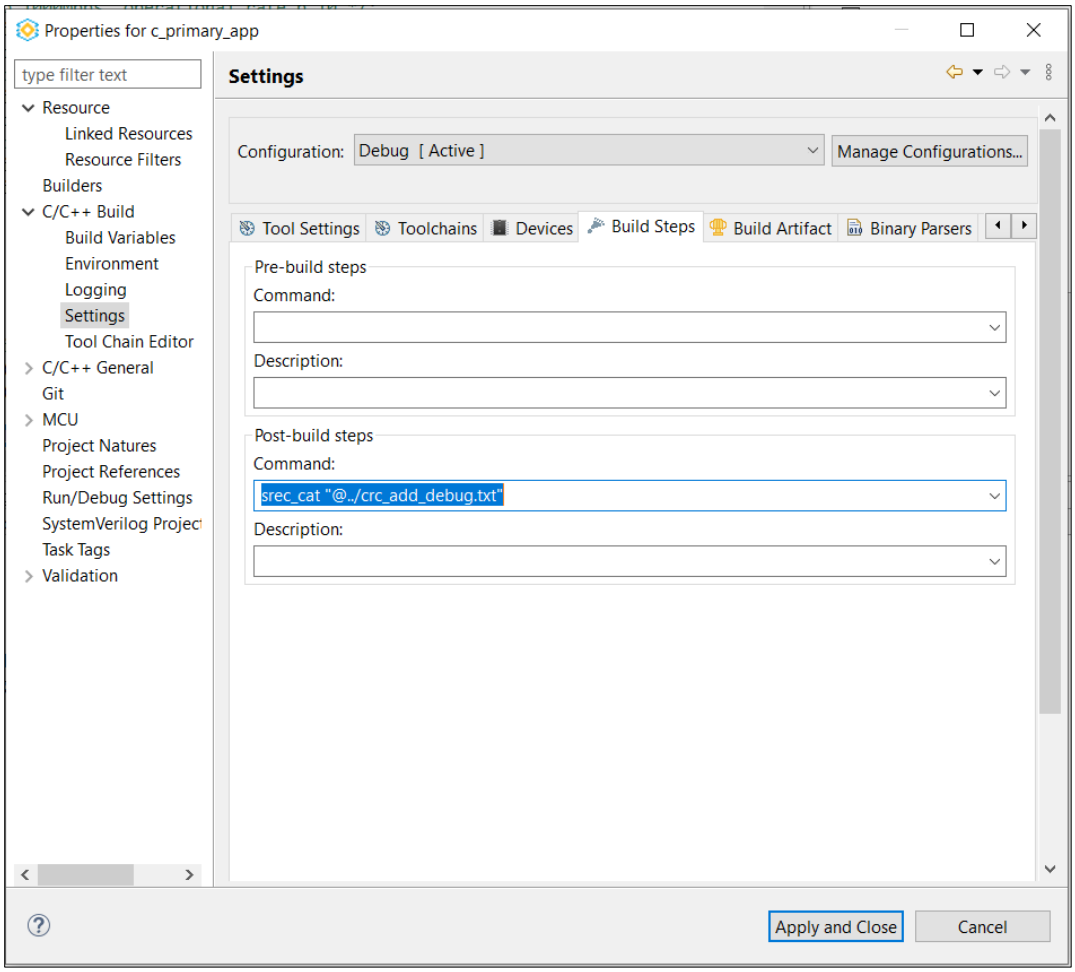


Figure 7.48. Adding Post Build Step for FreeRTOS Application CRC Binary Append (Linux)

16. For Linux system, update the path within the crc script to match Linux path format.

```
# srec_cat command file to add the CRC and produce application file to be flashed
# Usage: srec_cat @filename

#first: create CRC checksum
../Debug/c_golden_app.bin -Binary
-fill 0xFF 0x0000 0x40000          # fill code area with 0xff
-crop 0x0000 0x3fffe               # just keep code area for CRC calculation below
-CRC16_Big_Endian 0x3fffe -CCITT   # calculate big endian CCITT CRC16 at given address
-crop 0x3fffe 0x40000              # keep the CRC itself

#second: add application file
../Debug/c_golden_app.bin -Binary
-fill 0xFF 0x0000 0x3fffe          # fill code area with 0xff

# generate a Binary file
-o ../Debug/c_golden_appcrc.bin -Binary
```

Figure 7.49. Update the CRC Script to Match Linux Path Format

17. Click **Apply** and **Close**.
18. Right-click on **c_primary_app** and click on **Build Project**.

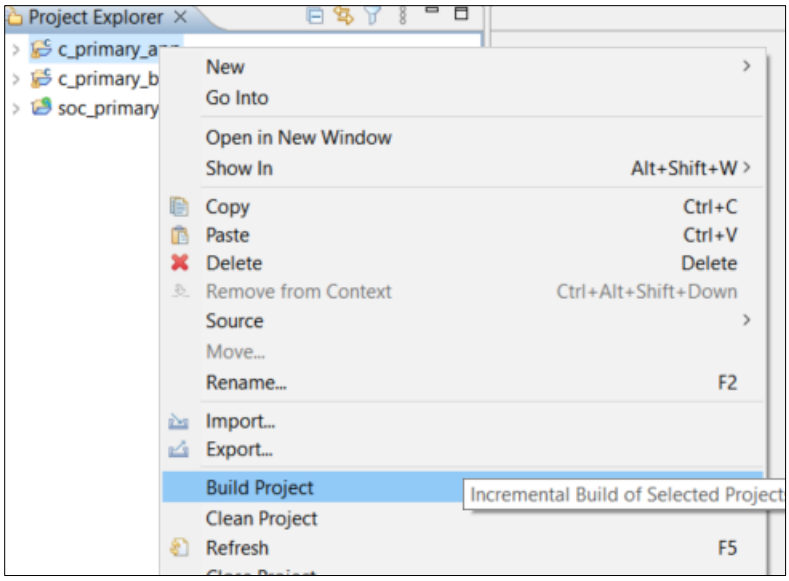


Figure 7.50. Build c_primary_app C/C++ Project

19. The console output is displayed as shown in Figure 7.51.

```
Invoking: GNU RISC-V Cross Create Listing
riscv-none-embed-objdump --source --all-headers --demangle --line-numbers --wide "c_primary_app.elf" > "c_primary_app.lst"
Invoking: GNU RISC-V Cross Print Size
riscv-none-embed-size --format=berkeley "c_primary_app.elf"
Invoking: Lattice Create Memory Deployment
riscv-none-embed-objcopy -O binary --gap-fill 0 "c_primary_app.elf" "c_primary_app.bin"; srec_cat "c_primary_app.bin" -Binary -byte-swap 4 -DISable Header -Output "c_primary_app.mem" -MEM 32
   text    data     bss     dec     hex filename
  84852     104   67292  152248   252b8 c_primary_app.elf
Finished building: c_primary_app.siz

Finished building: c_primary_app.lst

Finished building: c_primary_app.mem

srec_cat.exe "@..\crc_add_debug.txt"


15:13:27 Build Finished. 0 errors, 36 warnings. (took 2m:427ms)
```

Figure 7.51. FreeRTOS App Build Project Console Output (Note: Warnings can be ignored)

20. This creates the debug folder and the following file with **c_primary_appcrc.bin** binary with CRC as shown in Figure 7.52.

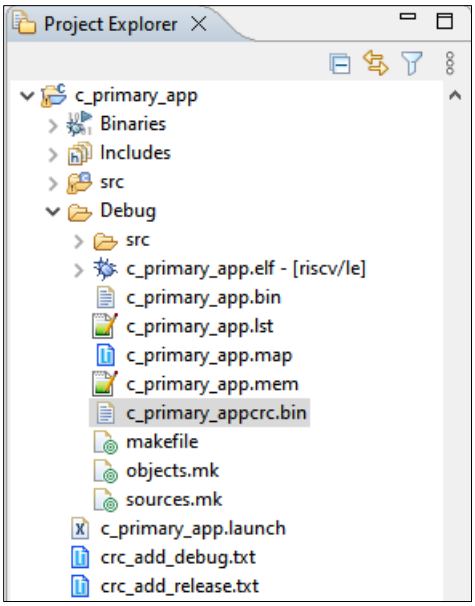


Figure 7.52.FreeRTOS App Binaries Created with CRC

21. For programming into the flash and confirming both primary bootloader and FreeRTOS application projects are functioning correctly, refer to the Programming Standalone Golden or Primary GSRD Bitstream and Application Software section.
22. To create the following files with **c_golden_appcrc.bin** binary, set the **#define _GOLDEN_BUILD_** in the *main.c* file to build for golden bootloader and golden app binaries.

```
54 /* Set for GOLDEN OR PRIMARY build */
55 #define _GOLDEN_BUILD_
56
```

Figure 7.53.Golden Build Define – Set _GOLDEN_BUILD_ for Golden Build in main.c

23. Repeat steps 1 to 21 to create golden bootloader and golden app project.

7.6. Generating the Multi-Boot MCS File

To generate the multi-boot MCS file, perform the following steps:

Note: Follow these steps only when you have or re-created both Golden and Primary bitstreams.

1. Launch the Lattice Radiant Programmer tool from Lattice Radiant as shown in Figure 7.54.

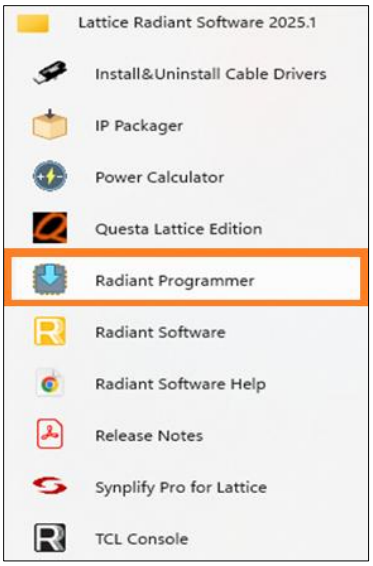


Figure 7.54. Launch Radiant Programmer from Windows Start

2. Provide the location where you want to store the programmer *.xcf* file. Click **OK**.

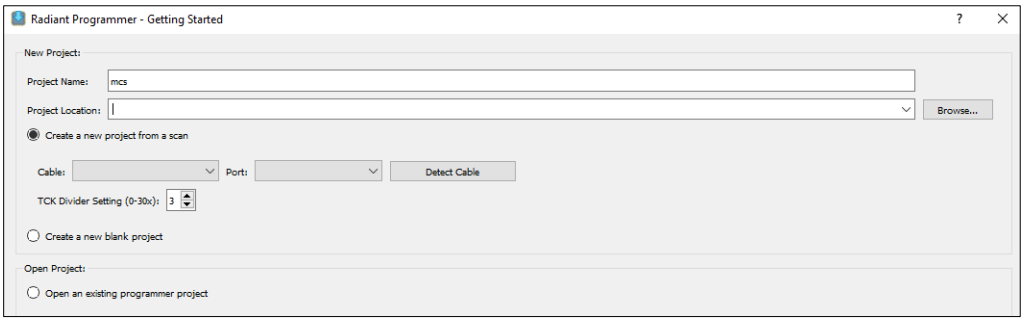


Figure 7.55. Radiant Programmer Getting Started Window

Note: This displays the error message shown in Figure 7.56 since there is no hardware board connected. This error message can be ignored.

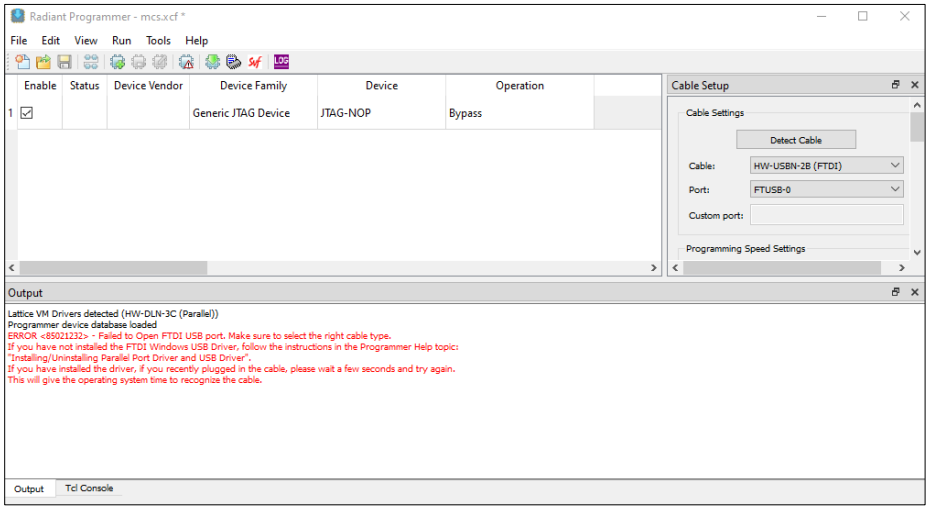


Figure 7.56. Error if No Board is Connected

3. Click **Tools > Deployment Tool**.

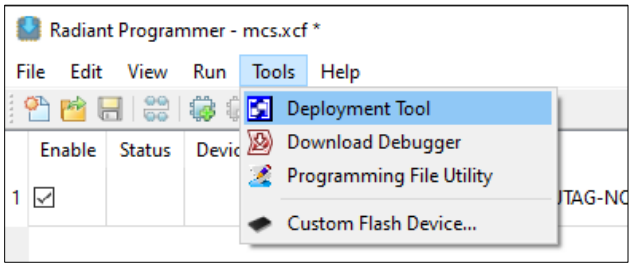


Figure 7.57. Open Deployment Tool from Radiant Programmer

4. This opens the Deployment Tool window as shown in Figure 7.58.

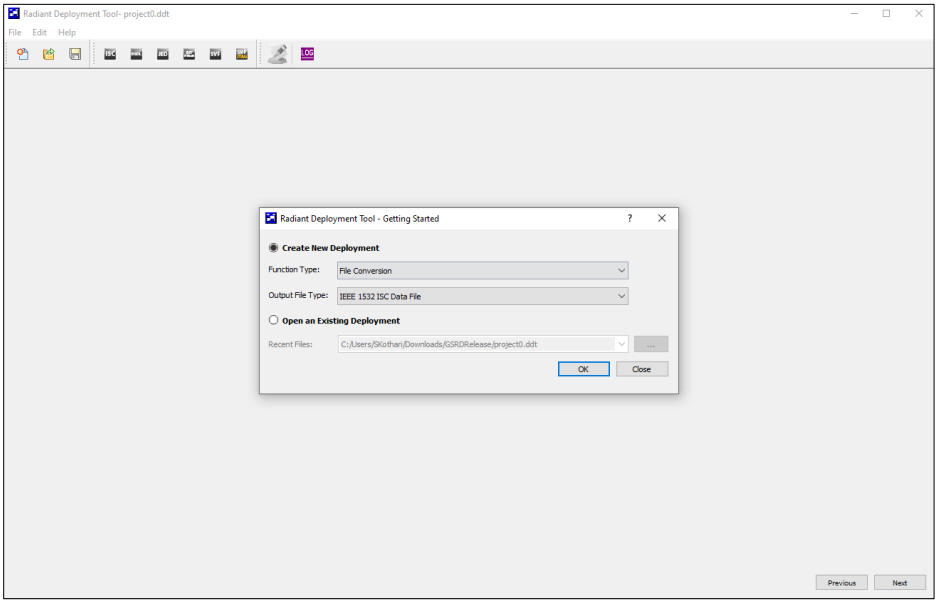


Figure 7.58. Deployment Tool Start Window

5. Apply the settings as shown in Figure 7.59 and click **OK**.

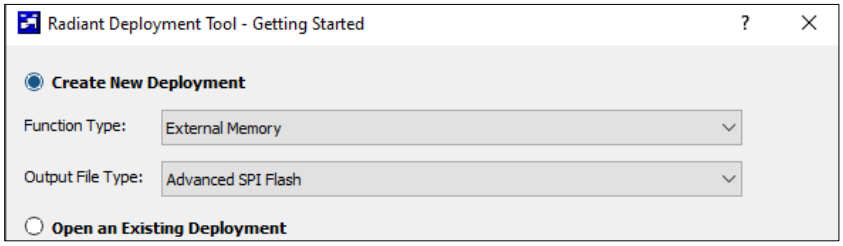


Figure 7.59. Options for Creating New Deployment

6. In **Step 1 of 4: Select the Input File(s)** window, as shown in Figure 7.60:
   a. Click the **File Name field** to browse and select the primary *.bit* from your primary soc project.
   b. The **Device Family** and **Device** fields auto-populate based on the bitstream *.bit* file selected.
   c. Click **Next**.

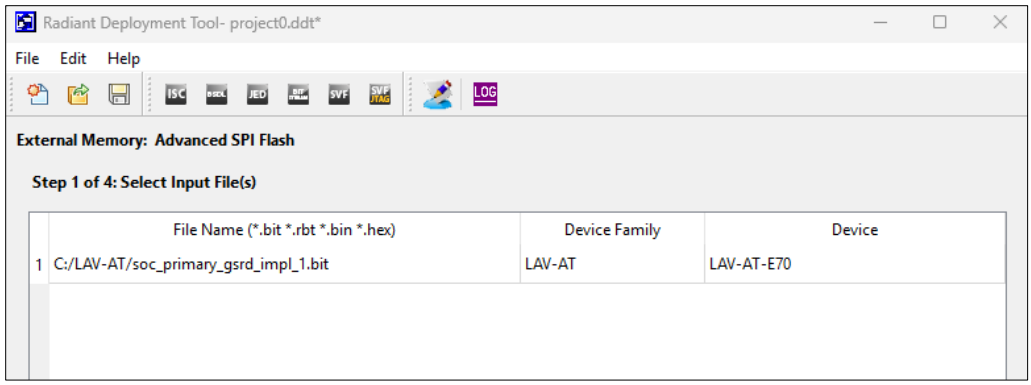
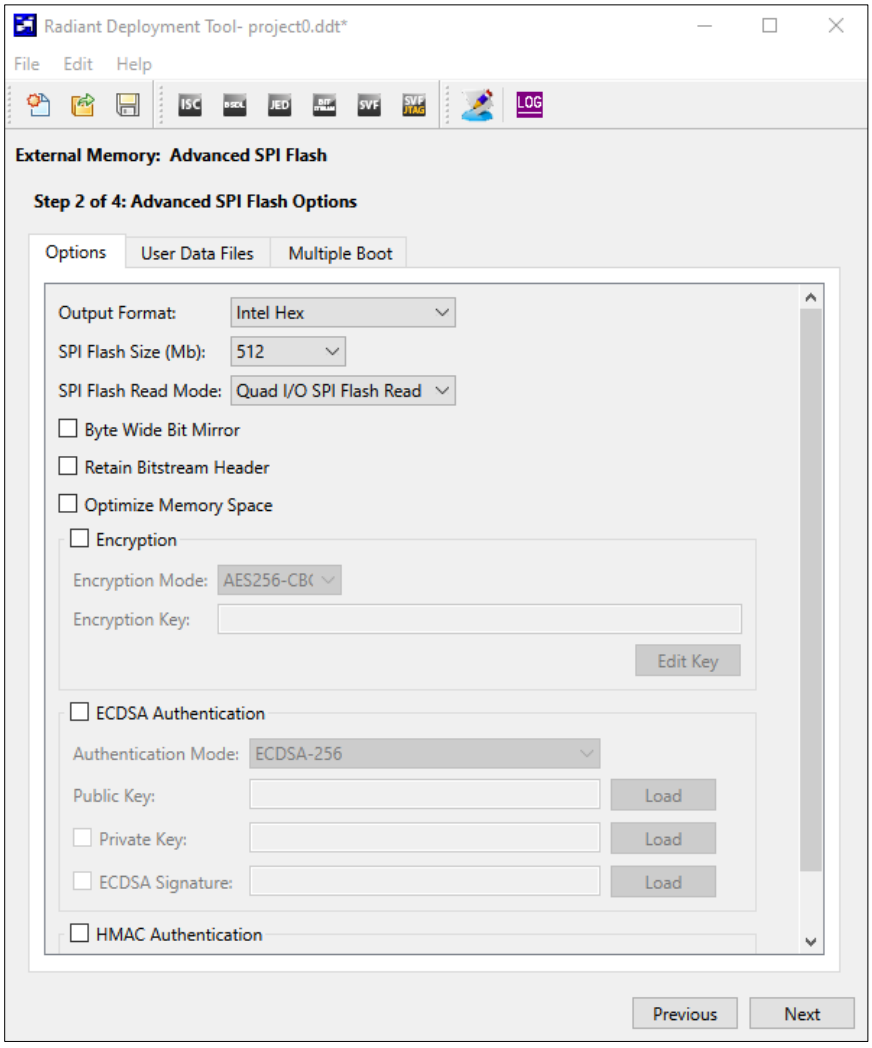


Figure 7.60. External Memory Step 1 of 4: Select Input Files

7. In **Step 2 of 4: Advanced SPI Flash Options**, select the fields as shown in Figure 7.61.
   a. Under **Options** tab, choose Output Format as **Intel Hex** and SPI Flash Size (Mb) as **512**.
   b. Select **Quad I/O SPI Flash Read** from the SPI Flash Read Mode dropdown menu.



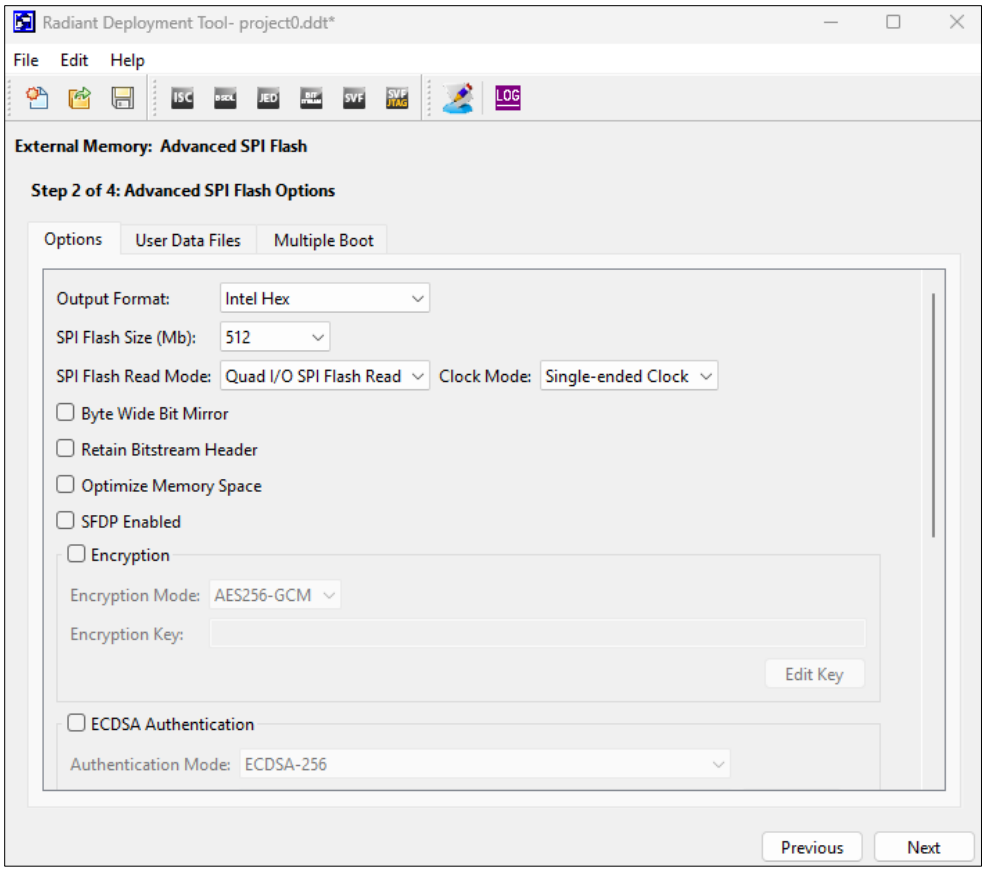


Figure 7.61. External Memory Step 2 of 4: Select Options

c. No changes needed in **User Data Files** section.

d. Under Multiple Boot tab.

   i. Select the **Multiple Boot** option.

   ii. Select **Number of Alternate Patterns** as 1.

   iii. Under the **Golden Pattern**, browse and select the golden SoC bitstream (that is *soc_golden_gsrd_impl_1.bit*) file. The **Starting Address** of **Golden Pattern** is automatically assigned.

   iv. Under **Alternate Pattern 1** field, click on the browse button and select the golden SoC bitstream. The **Starting Address** of this pattern is automatically assigned. You can change it by clicking on the drop-down menu. This is the pattern loaded during an event of next PROGRAMN/REFRESH or soft reset.

   v. Click **Next**.

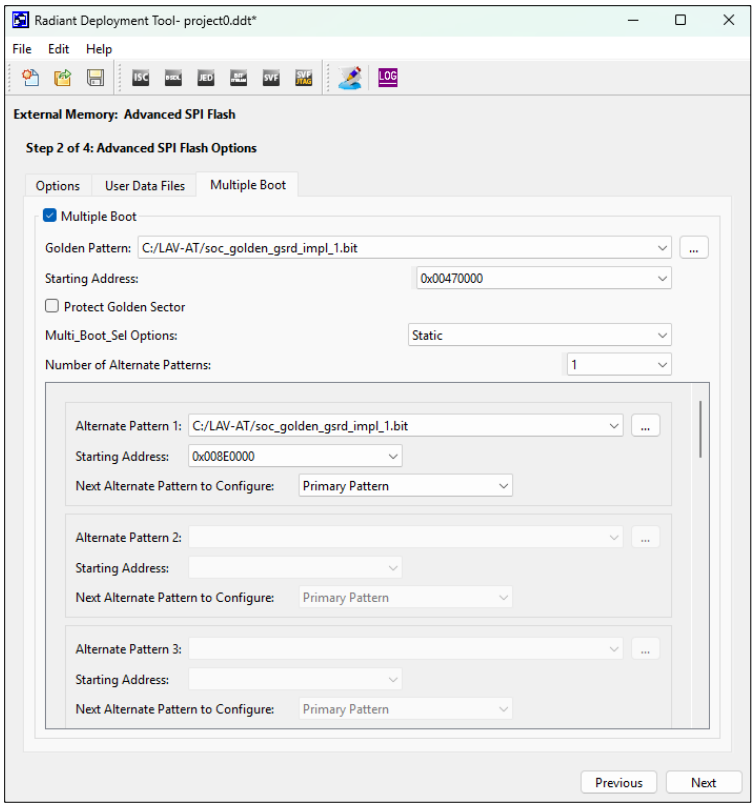


Figure 7.62. External Memory Step 2 of 4: Multi-Boot

8. In **Step 3 of 4: Select the Output File(s)** as shown in Figure 7.63. Choose the location on your machine to generate an *.mcs* file.

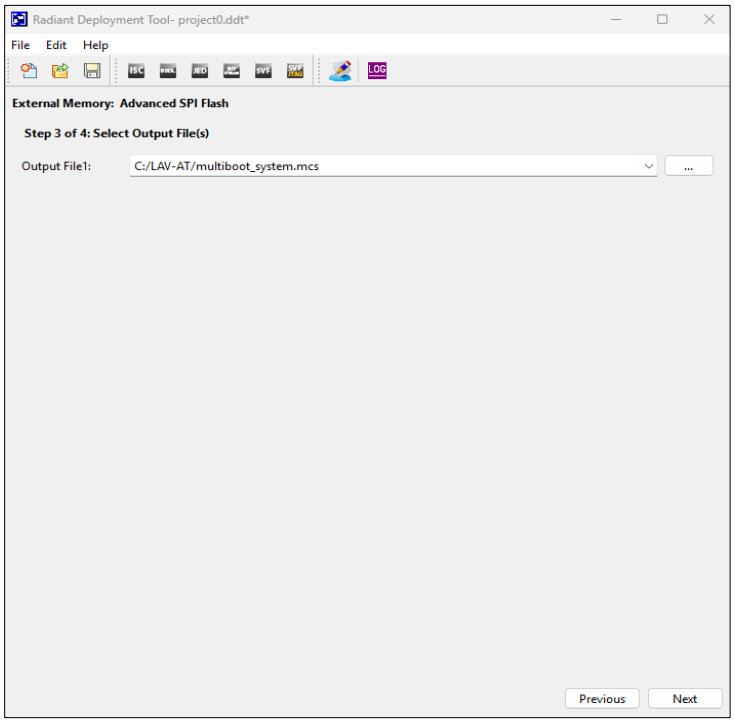


Figure 7.63. External Memory Step 3 of 4: Select Output File(s)

9. Click **Next**.
10. In **Step 4 of 4: Generate Deployment**, click **Generate** at the bottom right corner as shown in Figure 7.64.

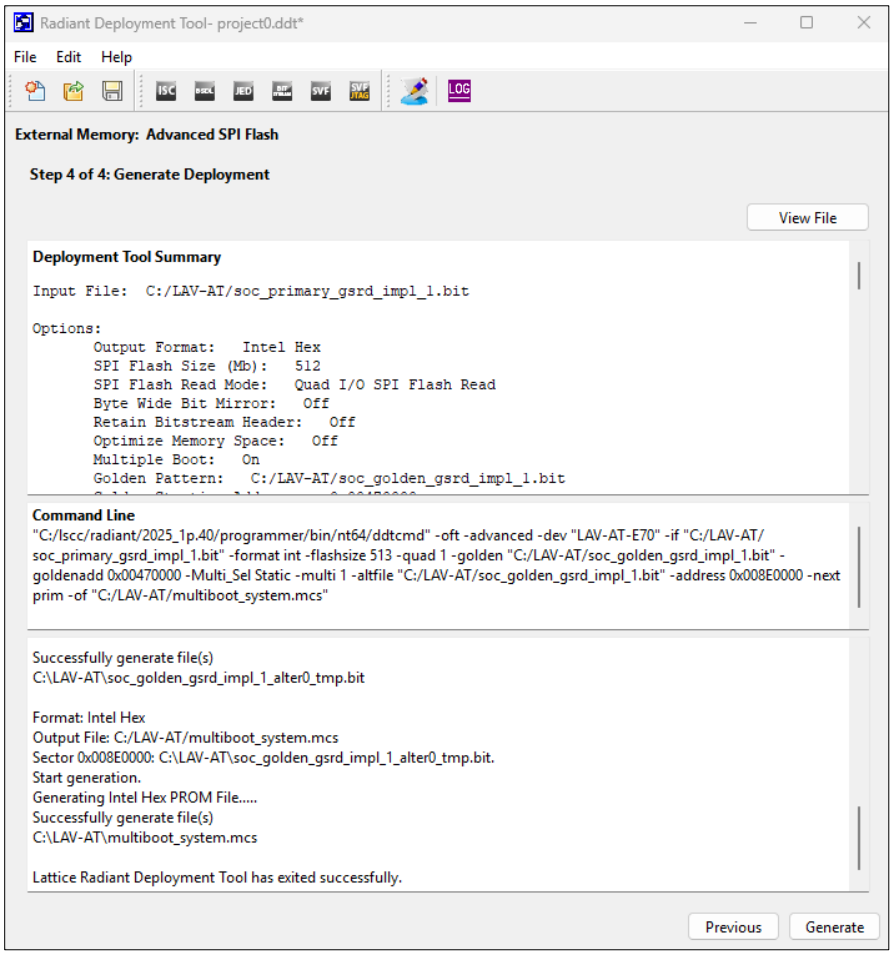


Figure 7.64. External Memory Step 4 of 4: General Development

11. Check for the following output as shown in Figure 7.65.

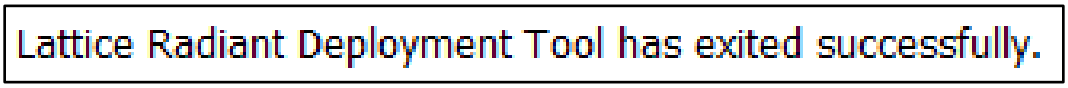


Figure 7.65. MCS File Generated Successfully

12. The generated final .mcs file is now ready to be programmed into the external flash using the Radiant Programmer.
13. Close the Deployment Tool window.
14. For programming into the flash and confirming the MCS file is built correctly, refer to the Programming Standalone Golden or Primary GSRD Bitstream and Application Software section.

7.7. Setup Host Machine for Ping Test (Windows)

The setup is based on the setup on Windows 11 Host Machine to the evaluation board. For Linux-based machine, refer to the setup procedure in the Linux Setup IP Address.

1. Search for **Ethernet settings** from the Windows Menu.

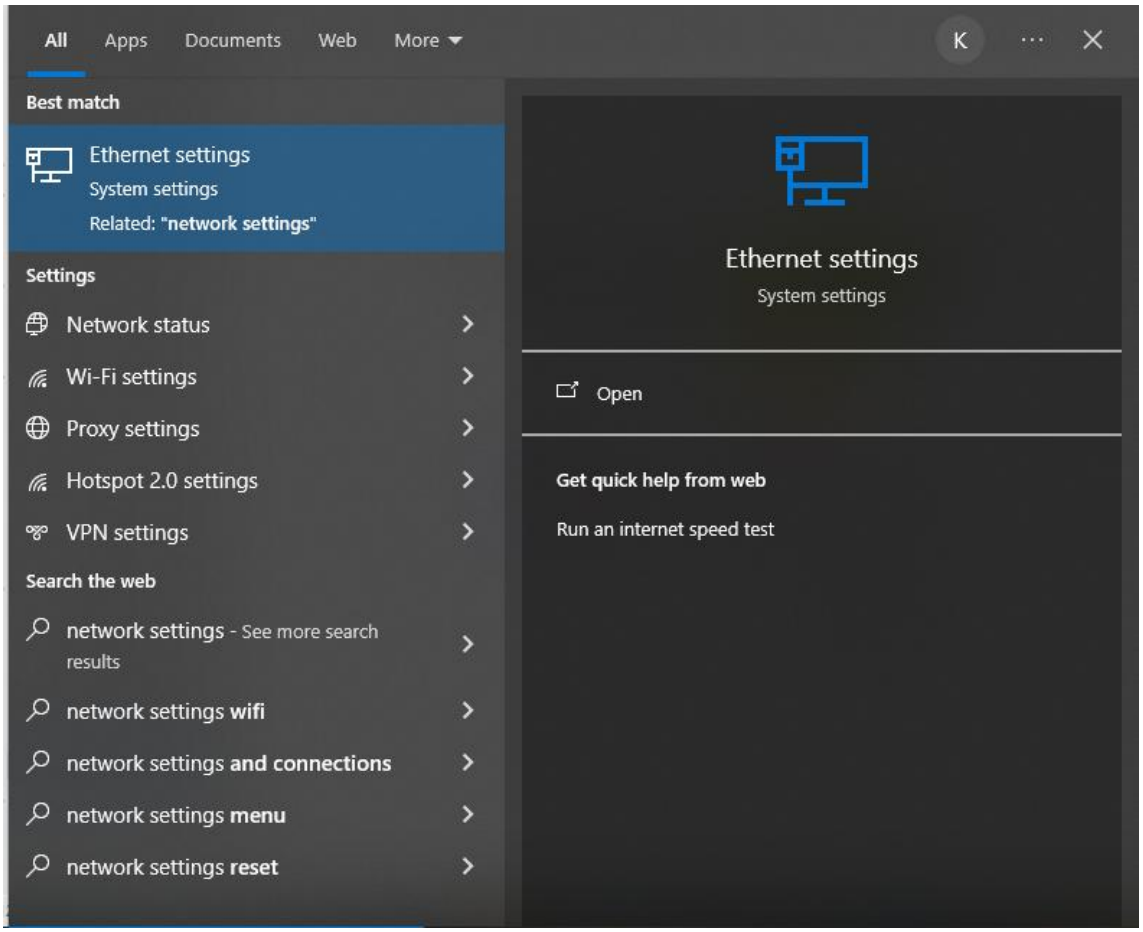


Figure 7.66. Configure Ethernet Settings

2. Select the **Ethernet** on the left panel and identify the Ethernet port connected to the evaluation board.

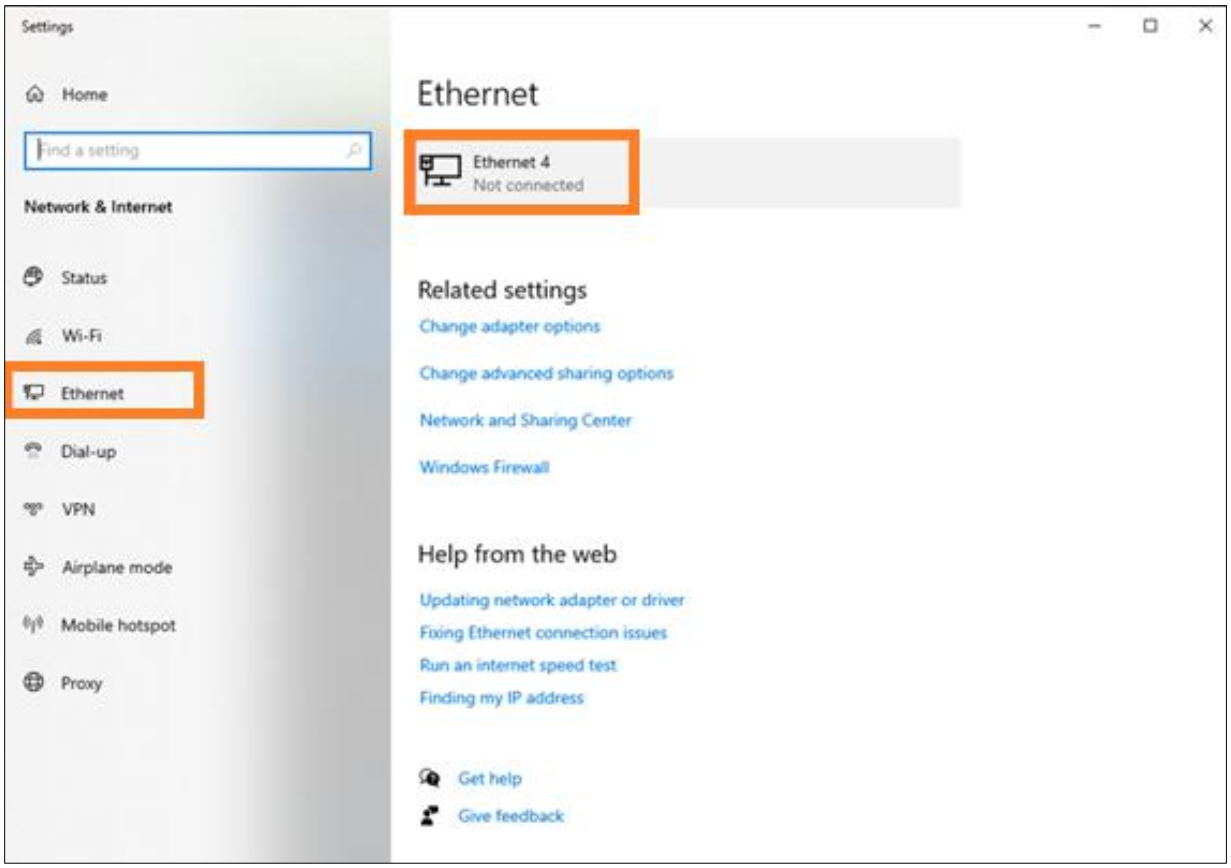


Figure 7.67. Select Ethernet Port

3. Click the **Ethernet** port connected to the evaluation port and click **Edit** at the **IP settings** section.
   Note: The status of **Ethernet 4** is **Not connected** because this port has not been configured.

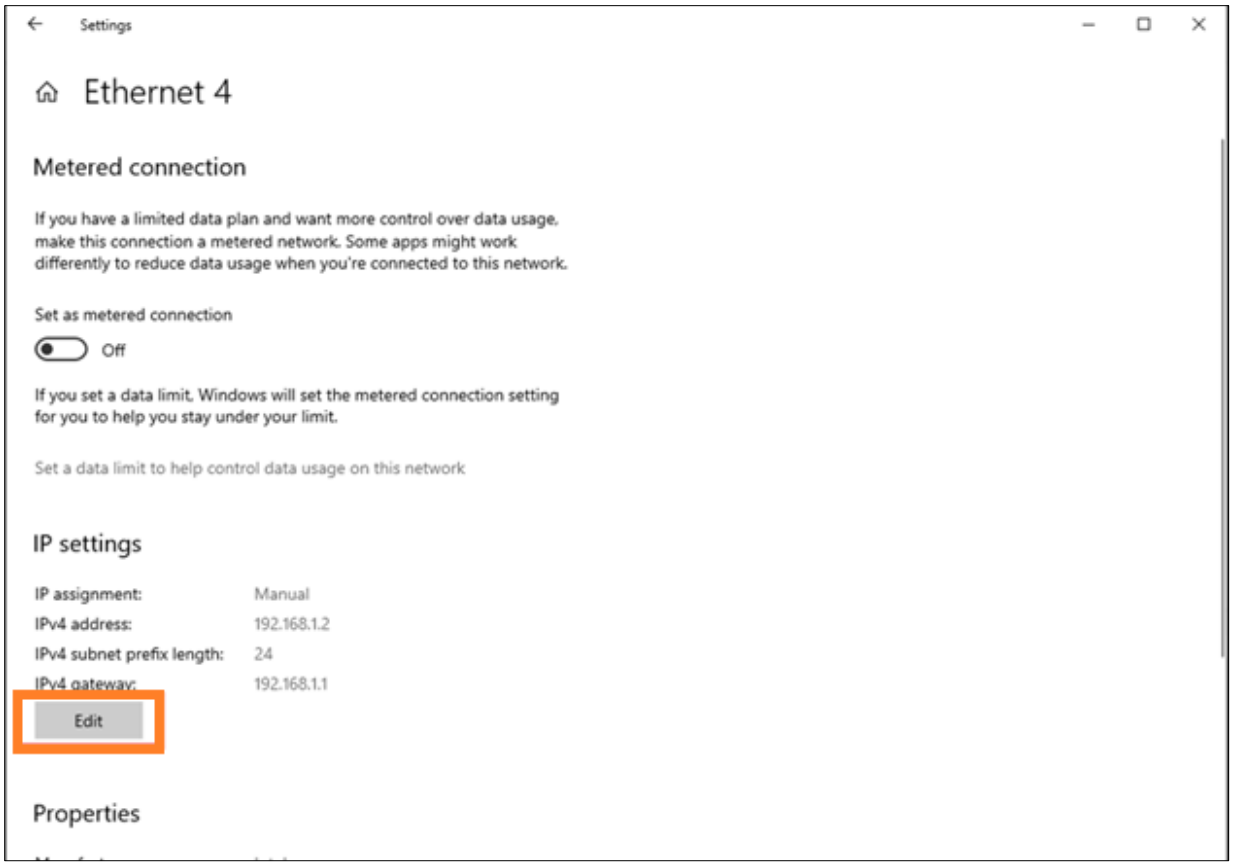


Figure 7.68. Edit IP Settings

4. Set up the IP address, Subnet prefix length, and Gateway. Click **Save**.
   - **Edit IP settings:** Manual
   - **IP address**: 192.168.1.2
   - **Subnet prefix length**: 24
   - **Gateway**: 192.168.1.1
   - **Preferred DNS** (required for Windows 11): 192.168.1.1

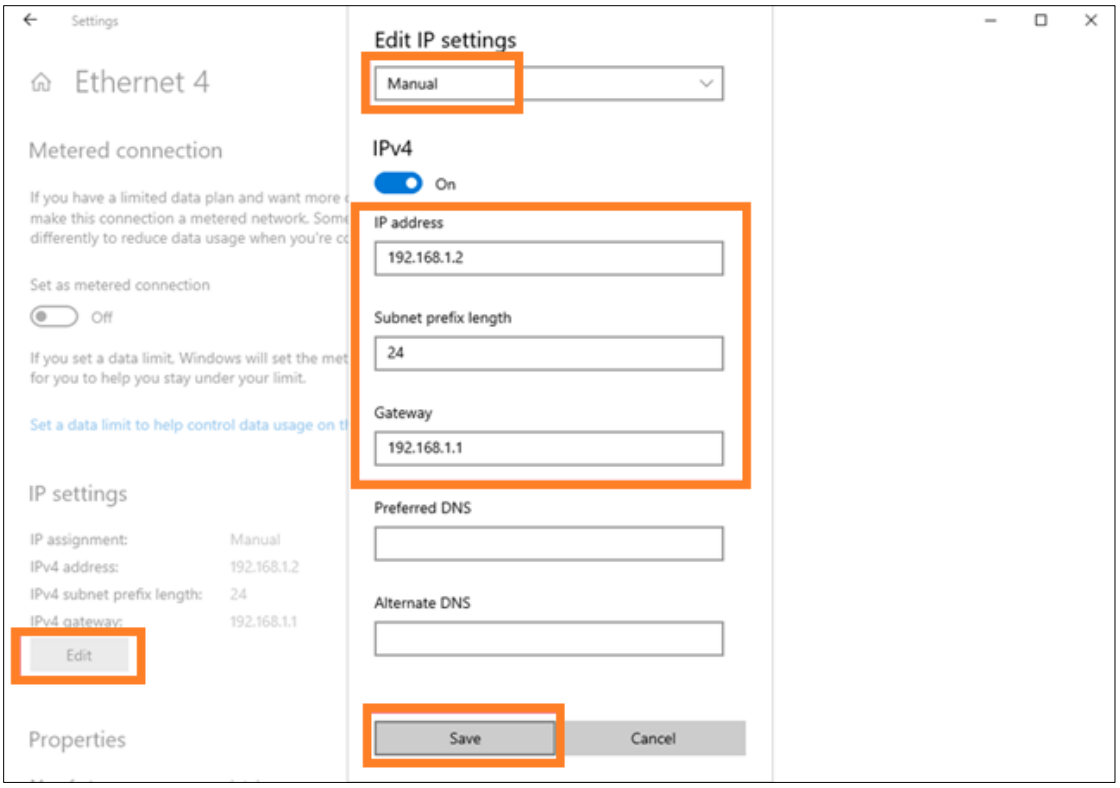


Figure 7.69. Set the IP Settings to Manual, IP Address, Subnet Prefix Length, and Gateway

5. Once the Ethernet settings are set up, open **Command Prompt** on the host PC.
6. Type **ipconfig** at the **Command Prompt** to check the connection between the host PC and evaluation board as shown in Figure 7.70.

```
Ethernet adapter Ethernet 4:

   Connection-specific DNS Suffix  . :
   Description . . . . . . . . . . . : Intel(R) Ethernet Connection (23) I219-LM
   Physical Address. . . . . . . . . : FC-5C-EE-B7-7E-7D
   DHCP Enabled. . . . . . . . . . . : No
   Autoconfiguration Enabled . . . . : Yes
   IPv4 Address. . . . . . . . . . . : 192.168.1.2(Preferred)
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1
   NetBIOS over Tcpip. . . . . . . . : Enabled
```

Figure 7.70. Check Ethernet Connection

7. Run the ping command: **ping 192.168.1.4**. A successful ping to the evaluation board should indicate the number of packets sent equal to the number of packets received with zero packet loss.

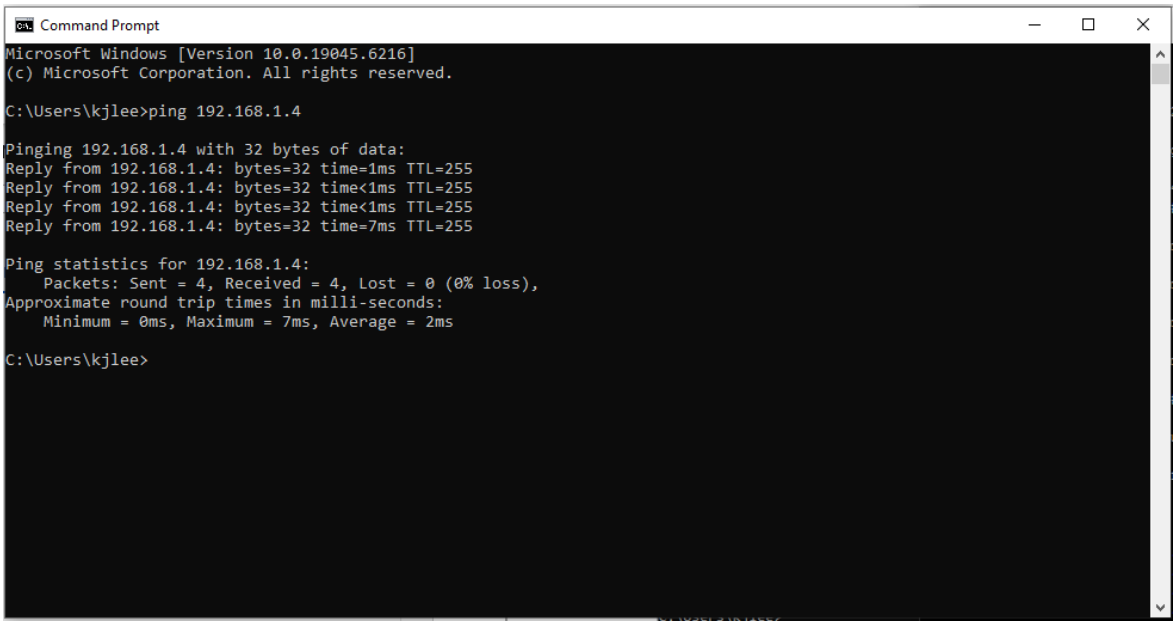


Figure 7.71. Ping the Device

8. Check the UART terminal on the evaluation board. Once the ping packets are received, dots are printed on the UART terminal.

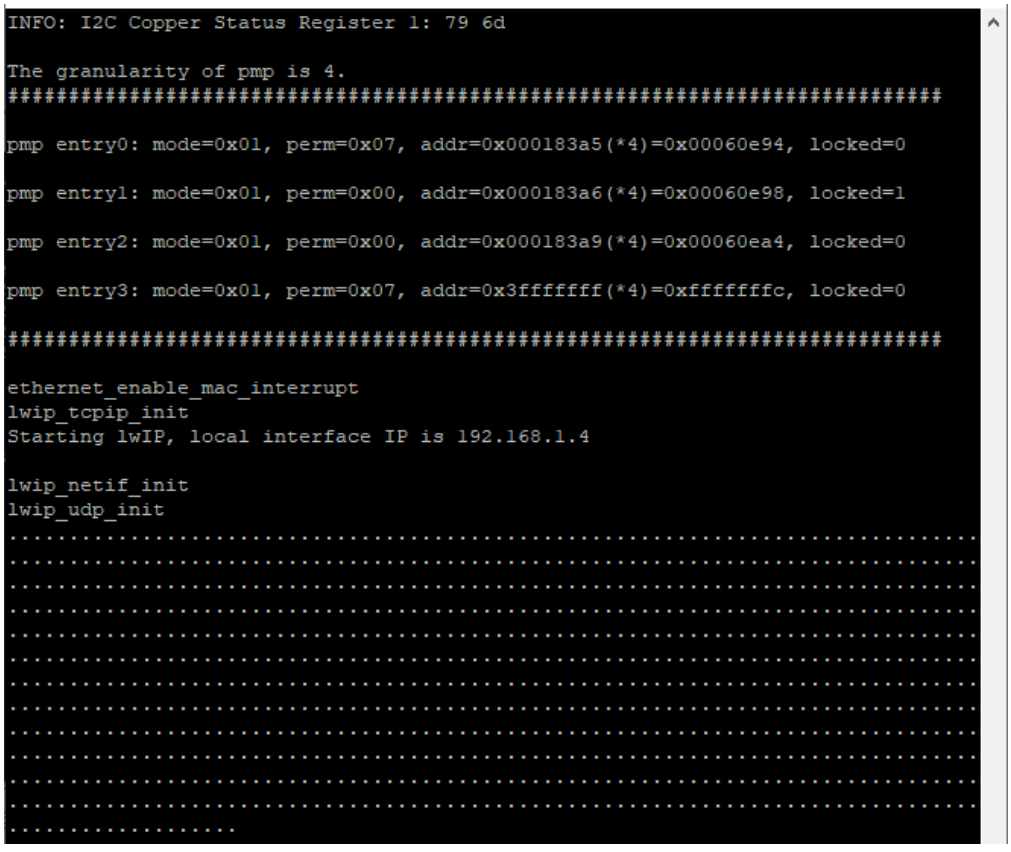


Figure 7.72. UART Terminal Showing the Printout When Ping Packet Received

8. Customizing the GSRD

This section describes the customization that can be applied to this reference design. The hardware related modifications are made by using Propel Builder, since it is the main design entry tool. The software-related modifications are made by using Propel SDK.

8.1. Adding Component to the GSRD

This reference design can be used as a base design to add components or IPs that your project requires. You can perform this in Propel Builder using the Schematic view. The following procedure shows the design flow in general.

8.1.1. Hardware Flow

To add a component or IP:

1. In **Propel Builder**, select and add IP from the **IP Catalog** window. Complete the IP configuration using the wizard and generate.

   Note: If you need to create a custom IP, refer to Lattice IP Packager 2025.1 (FPGA-UG-02236).
2. Connect the newly added IP to the RISC-V CPU or other IPs in the system. There are three primary interface types:
   - AXI4 or AXI-lite – Add new Manager/Subordinate interface in system_ic_inst (depending on the interface type on the new IP). Connect the newly added IP to the newly added interface on the interconnect.
   - APB – Add new Requestor/Completer interface in system_apb_ic_inst. Connect the newly added IP to the newly added interface on the interconnect.
   - AHB-Lite – Add new Manager/Subordinate interface in system_ic_inst. Since the newly added IP has AHB-Lite interface, you need to add AXI4 to AHB-Lite Bridge IP in between.
3. Connect the clock and reset signals, and data buses of the newly added IP.
4. Export the I/O from newly added IP to top level module (if applicable).
5. Go to **Address** view to assign the base address for the newly added IP.
6. If you made changes to the bootloader, follow these steps to update the System Memory's initialization file. Otherwise, skip this step.
   a. Select the System Memory instance from the **Design View** in the left column and it highlights the system_boot_mem_inst.
   b. Double-click on the highlighted component shown in the schematic and it opens the **Module/IP Block Wizard**.
   c. Click on the three dots at the **Initialization File's Value** field to locate and select your bootloader file.
   d. Click **Generate** at the bottom-right corner of the **Module/IP Block Wizard** as shown in Figure 8.1.

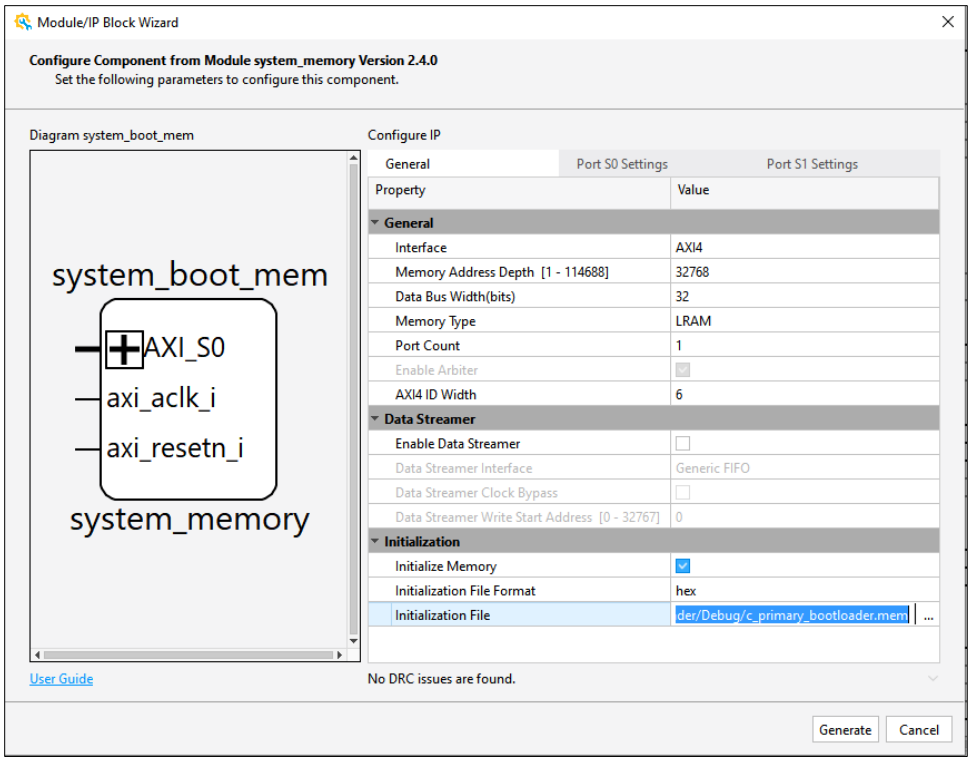


Figure 8.1. Bootloader File Updated in System Memory

7. Upon completion, select **Design > Validate Design** and make sure no DRC error.
8. Select **Design > Generate > Generate** to update the SoC design.
9. In **Lattice Radiant**, assign pins for the newly added IP (if applicable).
10. Click **Export Files** to generate the updated bitstream.

8.1.2. Software Flow

If the newly added IP contains software driver, update the Board Support Package (BSP) in the software project. To update the BSP, perform the following:

1. In Propel SDK, right-click on the software project that you are working on and select **Update Lattice C/C++ Project**. In the **Update System and BSP** dialog box, you may observe a **Directory is not correct!** error as shown in Figure 8.2. This is because the software project is created in another PC with a different system environment path when the project is imported to Propel SDK.

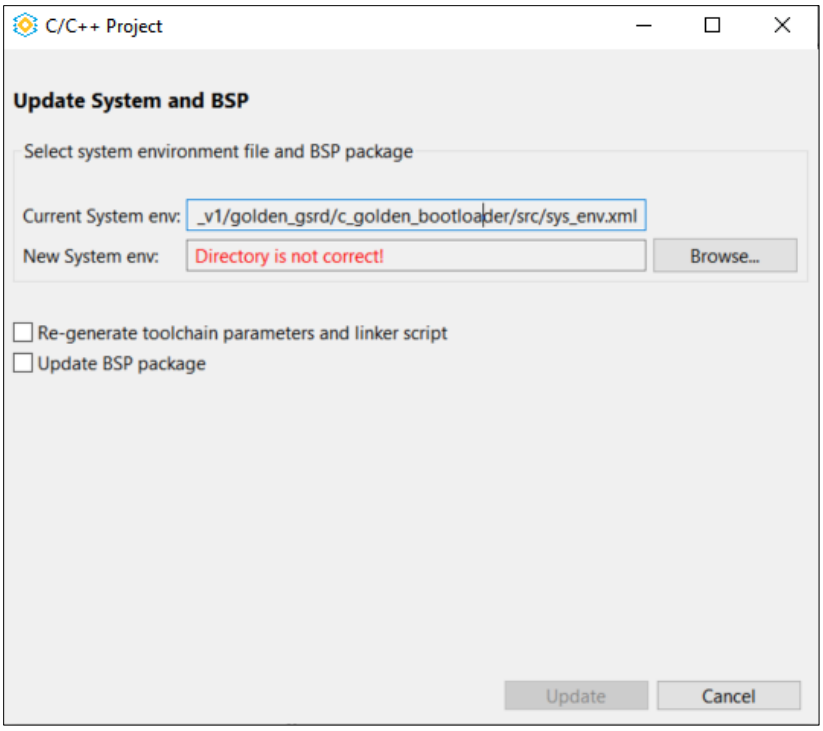


Figure 8.2. New System ENV Error

2. Click the **Browse** button to navigate to the new system env path in your PC.

   For example: <my_work_dir>/sge/sys_env.xml

   Note: This step updates the software project to point to the Propel Builder system env file located on your PC. The correct path is displayed instead of the previous error.
3. Select the **Update BSP package** box. This shows the newly added IP driver and version available for update.

   Note: Do NOT select **Re-generate toolchain parameters and linker script** box. The software projects provided in this reference design contain modified linker script. Selecting this option overwrites the modifications.
4. Click **Update** to complete the process.
5. Build the software project to obtain updated executable files (.elf, .mem, and .bin).

8.2. Using ECO Editor

During the initial design phase when the bootloader code is being developed, you may need to update the bootloader more frequently. This requires dynamic updating of bootloader file. Typically, the bootloader is loaded into the System Memory with *.mem* file during the bit file generation phase. The Engineering Change Order (ECO) editor tool allows you to update such *c_bootloader.mem* dynamically without requiring the entire Radiant flow to be re-run.

The following example demonstrates how to change/update the *.mem* file dynamically.

1. Click on **Tool > ECO Editor** or click on below icon. It opens the ECO Editor windows.

Figure 8.3. ECO Editor Icon in Radiant Software

2. Click on Memory Initialization tab.

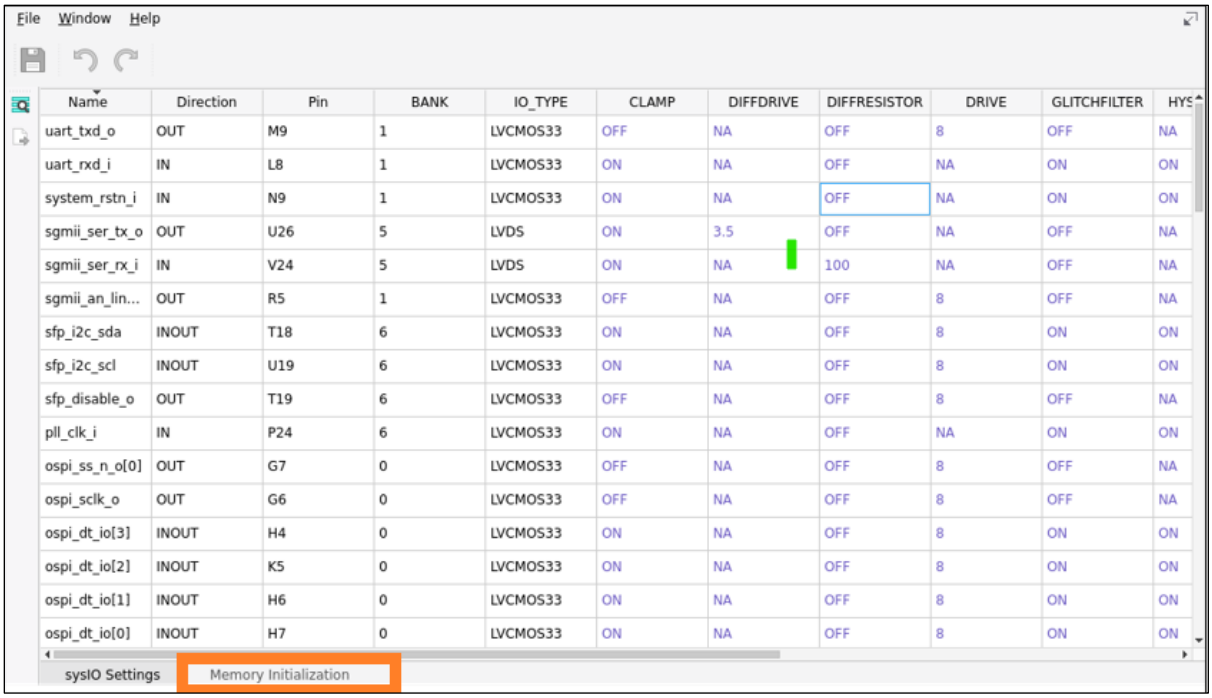


Figure 8.4. ECO Editor sysIO Settings Tab

3. Search for the *system_boot_mem* instance.

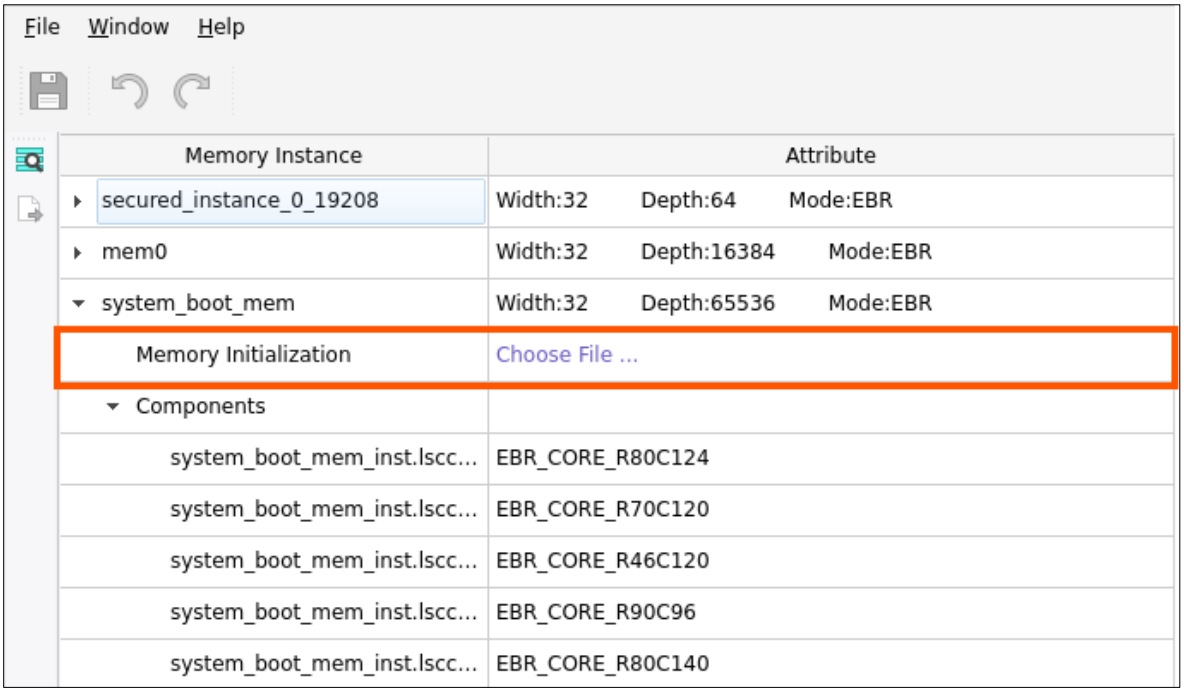


Figure 8.5. ECO Editor Memory Initialization Tab

4. Click on **Choose File …** to browse for *c_booloader.mem* file.

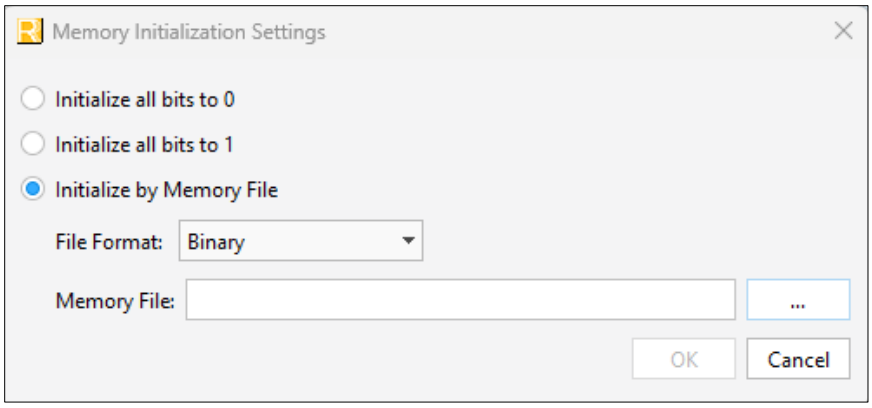


Figure 8.6. Select bootloader in Memory Initialization Settings

5. After *c_bootloader.mem* is loaded in the **Memory File** field, change **File format** to **Hexadecimal**. Click **OK**.

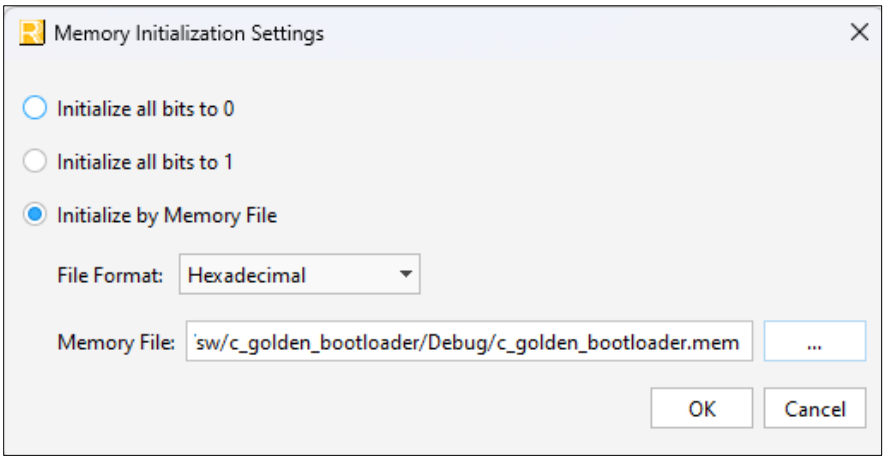


Figure 8.7. Change file format in Memory Initialization Settings

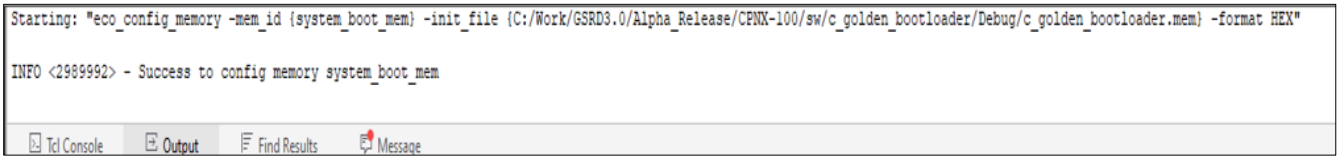


Figure 8.8. Updated System Memory Content

6. Click **Save**.

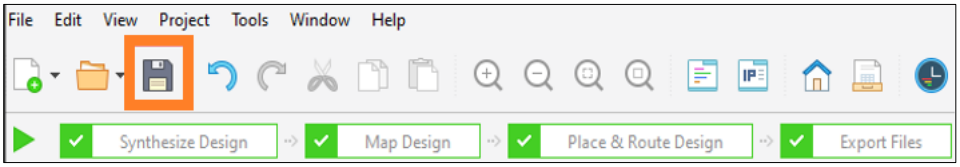


Figure 8.9. Save Icon

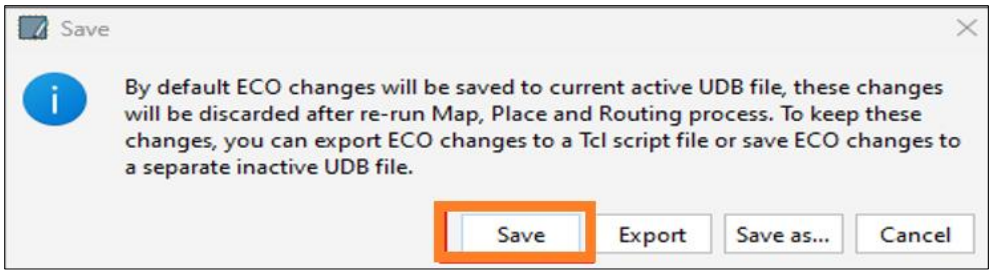


Figure 8.10. Save ECO Changes

7. After saving the project, hit the **Run** icon. The **Post Route Timing Analysis** and **Export Files** phases are re-run.

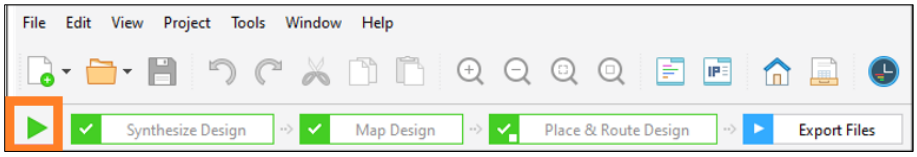


Figure 8.11. Re-run partial Radiant flow

8. The bitstream is re-generated with updated .mem file in the System Memory.

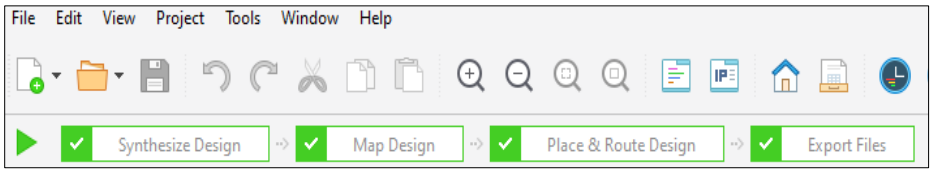


Figure 8.12. Completed Radiant flow

Figure 8.13. Bitstream is Re-Generated

8.3. IP Version Refresh without Structural Modification

The IPs in this reference design can be updated to the latest versions supported by the Lattice Propel Builder release. The following procedure describes the general flow for performing this update.

8.3.1. Hardware Flow

To update the IP:

1. In Propel Builder, select and add IP from the IP Catalog window.
2. Double-click the IP and complete the IP configuration using the wizard. Click **Generate**.
3. If the IP updates require changes to the bootloader, follow step 6 in the Hardware Flow section to update the System Memory's initialization file. Otherwise, skip this step.

8.3.2. Software Flow

If an updated IP includes a software driver, the Board Support Package (BSP) in the software project must also be updated. To update the BSP, follow the steps described in the Software Flow section.

The Avant-E70 Evaluation Board Rev D used in this reference design only supports Octal SPI in one lane. When updating this IP to the latest version, its default driver is configured to four lanes. Therefore, it is necessary to manually update the Octal SPI driver in the updated BSP package. The Octal SPI driver in the following software files needs to be updated.

- <my_work_dir>/c_golden_bootloader/src/bsp/driver/octal_spi_controller
- <my_work_dir>/c_golden_app/src/bsp/driver/octal_spi_controller
- <my_work_dir>/c_primary_bootloader/src/bsp/driver/octal_spi_controller
- <my_work_dir>/c_primary_app/src/bsp/driver/octal_spi_controller

To update the Octal SPI driver in the updated BSP package, perform the following steps:

1. In Lattice Propel SDK, navigate to **c_golden_bootloader/src/bsp/driver/octal_spi_controller**.
2. Open octal_spi_controller.c, search for **gencmd_quadspi_fast_read**, and replace the following x4 command with an x1 command under the SPI Flash device supported by the Avant-E70 device. For example, if the Avant-E70 device is equipped with a Winbond SPI Flash, update the command as shown in Figure 8.14. Repeat the same step in **gencmd_quadspi_page_program**, as shown in Figure 8.15.

   x4 Command:

```
op_param = (op_param_t){(handle->flash_addr_mode) ? FLASH_CMD_4IOFR4BYTE :
FLASH_CMD_4IOFR , SPIX8_IO_X1, SPIX8_IO_X4, SPIX8_IO_X4, 0, 0, 0, 6};
x1 Command:
op_param = (op_param_t){FLASH_CMD_READ, SPIX8_IO_X1, SPIX8_IO_X1, SPIX8_IO_X1, 0, 0,
0, 0};
```

```
1858  unsigned char gencmd_quadspi_fast_read(spix8_ctl_handle_t *handle,
1859                                         unsigned int num_bytes,
1860                                         unsigned int *dat_buf,
1861                                         unsigned int start_addr)
1862  {
1863      unsigned char status = SUCCESS;
1864
1865      if (!handle)
1866          return FAILURE;
1867
1868      op_param_t op_param;
1869
1870  #if (!defined _MICRON_X4_FLASH_DEVICE_)
1871      if (handle->flash_addr_mode == FLASH_ADDR_MODE_32B)
1872          gencmd_flash_set_4byte_mode(handle,FLASH_ADDR_MODE_32B);
1873      else
1874          gencmd_flash_set_4byte_mode(handle,FLASH_ADDR_MODE_24B);
1875  #endif
1876
1877      /* Device-specific command and dummy cycles */
1878  #if defined(_WINBOND_X4_FLASH_DEVICE_) || defined(_MACRONIX_X4_FLASH_DEVICE_)
1879      //op_param = (op_param_t){(handle->flash_addr_mode) ? FLASH_CMD_4IOFR4BYTE : FLASH_CMD_4IOFR , SPIX8_IO_X1, SPIX8_IO_X4, SPIX8_IO_X4, 0, 0, 0, 6};
1880      op_param = (op_param_t){FLASH_CMD_READ, SPIX8_IO_X1, SPIX8_IO_X1, SPIX8_IO_X1, 0, 0, 0, 0};
1881  #elif defined(_MICRON_X4_FLASH_DEVICE_)
1882      op_param = (op_param_t){FLASH_CMD_4IOFR, SPIX8_IO_X4, SPIX8_IO_X4, SPIX8_IO_X4, 0, 0, 0, 10};
1883  #else
1884      op_param = (op_param_t){FLASH_CMD_FAST_READ, SPIX8_IO_X4, SPIX8_IO_X4, SPIX8_IO_X4, 0, 0, 0, 2};
1885  #endif
1886
1887      set_op_param(&op_param);
1888      status &= flash_read(handle, num_bytes, dat_buf, start_addr, SPIX8_GEN_CMD, 0);
1889
1890      return status;
1891  }
```

Figure 8.14. gencmd_quadspi_fast_read

```
unsigned char gencmd_quadspi_page_program(spix8_ctl_handle_t *handle, unsigned int num_bytes, unsigned int *dat_buf, unsigned int start_addr)
{
    unsigned char status = SUCCESS;

    if (!handle)
        return FAILURE;

    op_param_t params;

#if (!defined _MICRON_X4_FLASH_DEVICE_)
    if (handle->flash_addr_mode == FLASH_ADDR_MODE_32B)
        gencmd_flash_set_4byte_mode(handle,FLASH_ADDR_MODE_32B);
    else
        gencmd_flash_set_4byte_mode(handle,FLASH_ADDR_MODE_24B);
#endif

#if (defined _MACRONIX_X4_FLASH_DEVICE_)
    params = (op_param_t){(handle->flash_addr_mode) ? FLASH_CMD_4IPP4BYTE : FLASH_CMD_4IPP, SPIX8_IO_X1, SPIX8_IO_X4, SPIX8_IO_X4, 0, 0, 0, 0};
#elif (defined  _WINBOND_X4_FLASH_DEVICE_)
    //params = (op_param_t){(handle->flash_addr_mode) ? FLASH_CMD_4IPP4BYTE : FLASH_CMD_4IPP, SPIX8_IO_X1, SPIX8_IO_X1, SPIX8_IO_X4, 0, 0, 0, 0};
    params = (op_param_t){FLASH_CMD_PP, SPIX8_IO_X1, SPIX8_IO_X1, SPIX8_IO_X1, 0, 0, 0, 0};
#else
    params = (op_param_t){FLASH_CMD_4IPP, SPIX8_IO_X4, SPIX8_IO_X4, SPIX8_IO_X4, 0, 0, 0, 0};
#endif

    //gencmd_flash_set_quad_mode(handle, FLASH_QUADSPI_ENABLE);
    set_op_param(&params);
    status &= flash_program(handle, num_bytes, dat_buf, start_addr, SPIX8_GEN_CMD);

    return status;
}
```

Figure 8.15. gencmd_quadspi_page_program

3. Next, open octal_spi_controller.h and change the device to Winbond in the header, as shown in Figure 8.16.

```
50  ================================================================================ */
51  #ifndef OCTAL_SPI_CONTROLLER_H_
52  #define OCTAL_SPI_CONTROLLER_H_
53
54  //#define _MACRONIX_X4_FLASH_DEVICE_
55  #define _WINBOND_X4_FLASH_DEVICE_
```

Figure 8.16. Define Winbond Device

4. Apply the same steps to update the Octal SPI driver in the respective software files mentioned above.
5. Build the software project to obtain updated executable files (.elf, .mem, and .bin).

   Note: If the error appears when build the software project, refer to the Debugging the GSRD section for the detailed debugging instructions.

9. Building the RISC-V Zephyr

The Golden System Reference Design (GSRD) now supports Zephyr as an additional real-time operating system (RTOS). In this reference design, Zephyr RTOS serves as the RISC-V firmware component for the GSRD project. This section provides a comprehensive guide to configuring the Zephyr build environment, compiling the firmware, and deploying the Zephyr binary to the target device. The Zephyr setup and compilation steps are performed on an Ubuntu operating system, validated on Ubuntu 22.04 LTS and 24.02 LTS.

Below are the prerequisites:

- Lattice Propel 2025.1.1 64-bit for Linux installed in the Ubuntu OS.
- Working and successfully built Golden bootloader and bitstream performed in the Compiling and Running the Reference Design section.

9.1. Configuring the Zephyr Build Environment

To configure the Zephyr Build environment, perform the following steps:

1. Install the required dependencies:
```
sudo apt install --no-install-recommends git cmake ninja-build gperf \
ccache dfu-util device-tree-compiler wget \
python3-dev python3-pip python3-setuptools python3-tk python3-wheel xz-utils file \
make gcc gcc-multilib g++-multilib libsdl2-dev libmagic1 srecord
```
2. Verify the versions of the main dependencies installed on your system by entering:
```
cmake –version
python3 –versiondtc --version
```
3. Install Zephyr's additional Python dependencies in a Python virtual environment. Install Python venv package:
```
sudo apt install python3-venv
```
4. Create a directory named *zephyrproject* in the home directory. The folder hosts all Zephyr related project files.
```
mkdir ~/zephyrproject
```
5. Create a new virtual environment.
```
python3 -m venv ~/zephyrproject/.venv
```
6. Activate the virtual environment. Once activated, your shell is prefixed with .venv. The virtual environment can be deactivated at any time by running deactivate command.

   Note: Remember to activate the virtual environment every time you start working.
```
source ~/zephyrproject/.venv/bin/activate
```
7. Install west by running the command below.
```
pip install west
```
8. Clone Zephyr GSRD source code into the local machine:
   - GSRD Zephyr repository: git@github.com:LatticeSemi/Zephyr-Lattice.git
   - GSRD Zephyr branch: lattice_zephyr_v2025.01
```
git clone git@<github_link.git> -b <branch_name> ~/zephyrproject/Zephyr
```
9. Initialize the Zephyr project.
```
west init -l ~/zephyrproject/Zephyr
cd ~/zephyrproject
west update
```
10. Export a Zephyr CMake package. This allows CMake to automatically load boilerplate code required for building Zephyr applications.
```
west zephyr-export
```
11. The Zephyr's scripts/requirements.txt file declares additional Python dependencies. Install the files with pip.
```
pip install -r ~/zephyrproject/Zephyr/scripts/requirements.txt
```

9.2. Building the Zephyr

To build the Zephyr, perform the following steps:

1. Activate python virtual environment.
   ```
   source ~/zephyrproject/.venv/bin/activate
   ```
2. Set ZEPHYR_BASE path to the zephyr project directory:
   ```
   export ZEPHYR_BASE=<ZEPHYR_GIT_DIRECTORY>/Zephyr
   ```
3. Set the following environment variable for Zephyr cross-compilation:
   ```
   export ZEPHYR_TOOLCHAIN_VARIANT=cross-compile
   #Cross-compiler path can be found upon Propel installation with prefix
   export CROSS_COMPILE=<PROPEL_PATH>/lscc/propel/2025.1/sdk/riscv-none-embed-
   gcc/bin/riscv-none-embed-
   #Cross-compiler toolchain path
   export TOOLCHAIN_HOME=<PROPEL_PATH>/lscc/propel/2025.1/sdk/riscv-none-embed-gcc
   ```
4. To enable the device ping command application, refer to the Ping Test (Device to Host) section to enable in the defconfig.
5. Run the build command.
   ```
   west -v build -p always -b lattice_avant samples/hello_world
   ```
6. The output message as shown in Figure 9.1 is observed if the build is successful.

```
Memory region         Used Size  Region Size  %age Used
           RAM:      197252 B         2 GB      0.01%
       IDT_LIST:          0 GB         2 KB      0.00%
Generating files from /home/admin-ubuntu/zephyrproject/Zephyr/build/zephyr/zephyr.elf for board: lattice_avant
(.venv) admin-ubuntu@adminubuntu-desktop:~/zephyrproject/Zephyr$
```

Figure 9.1. Success Zephyr Build

7. Additionally, the zephyr_crc.bin file is generated in *~/zephyrproject/Zephyr/build/zephyr* directory. The file is used to program into the SPI flash device to run the Zephyr firmware.

```
(.venv) admin-ubuntu@adminubuntu-desktop:~/zephyrproject/Zephyr$ ls build/zephyr/
arch                edt.pickle         lib                        snippets_generated.cmake  zephyr.dts.pre
boards              include            libzephyr.a                soc                       zephyr.elf
cmake               isr_tables.c       linker.cmd                 subsys                    zephyr_final.map
CMakeFiles          isr_tables_swi.ld  linker.cmd.dep              zephyr.bin                zephyr.map
cmake_install.cmake isr_tables_vt.ld   linker_zephyr_pre0.cmd     zephyr_crc.bin            zephyr_pre0.elf
drivers             kconfig            linker_zephyr_pre0.cmd.dep zephyr.dts                zephyr_pre0.map
dts.cmake           kernel             misc                       zephyr.dts.d              zephyr.stat
```

Figure 9.2. Generated zephyr_crc.bin

9.3. Running the Zephyr

1. To run Zephyr, program the *zephyr_crc.bin* into the SPI flash. Refer to the Programming Standalone Golden or Primary GSRD Bitstream and Application Software section for more details.
2. Flash the golden bitstream to CRAM.
3. Once the bootloader is booted, it jumps to Zephyr. Once Zephyr starts, it performs the GPIO test by toggling LED on/off.

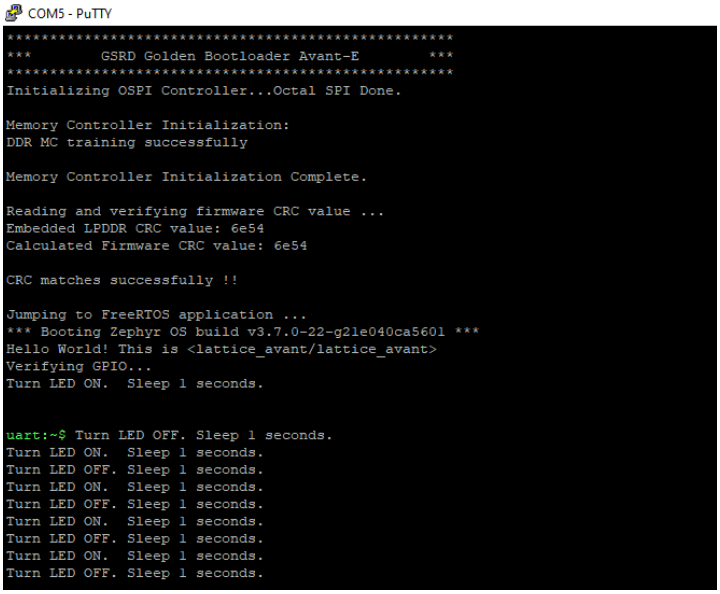


Figure 9.3. Zephyr Boot-Up

9.4. Read, Write, and Erase SPI Flash Device with the Zephyr Commands

Zephyr provides a UART terminal interface that allows you to execute various commands, including operations on SPI flash memory. This section explains how to use Zephyr commands to read, write, and erase memory content on an SPI flash device.

Below are the supported flash commands in Zephyr:

- Read command:
  ```
  flash read <address> <length_in_hex>
  ```
- Write command:
  ```
  flash write <address> <hex data>
  ```
- Erase command:
  ```
  flash erase <address> < length_in_hex >
  ```

To read/write/erase the SPI flash device, perform the following:

1. To perform a flash read from SPI flash device offset at 0x2900000 for 48 bytes of data, run the sample command below.
   ```
   flash read ospi@c4000000 0x2900000 30
   ```
2. Write some content to the SPI flash device.
   ```
   flash write ospi@c4000000 0x2900000 11112222 33334444 55556666 77778888
   ```
3. Read back the content from SPI flash device to check if it is updated.
   ```
   flash read ospi@c4000000 0x2900000 30
   ```
4. Erase the content from SPI flash device.
   ```
   flash erase ospi@c4000000 0x2900000 30
   ```
5. Read back the flash offset to verify that it is correctly erased.
   ```
   flash read ospi@c4000000 0x2900000 3
   ```

```
uart:~$ flash read ospi@c4000000 0x2900000 30
02900000: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff |........ ........|
02900010: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff |........ ........|
02900020: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff |........ ........|

uart:~$ flash write ospi@c4000000 0x2900000 11112222 33334444 55556666 77778888
Write OK.
Verified.
uart:~$ flash read ospi@c4000000 0x2900000 30
02900000: 22 22 11 11 44 44 33 33  66 66 55 55 88 88 77 77 |""..DD33 ffUU..ww|
02900010: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff |........ ........|
02900020: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff |........ ........|

uart:~$ flash erase ospi@c4000000 0x2900000 30
Erase success.
uart:~$ flash read ospi@c4000000 0x2900000 30
02900000: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff |........ ........|
02900010: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff |........ ........|
02900020: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff |........ ........|
```

Figure 9.4. Zephyr Flash Read/Write/Erase Commands

9.5. Ping Test (Host to Device)

To run ping test triggered by host machine, refer to the Setup Host Machine for Ping Test (Windows) section. The host machine shows the number of ping packets sent and received to/forth from the device.

```
Command Prompt
C:\Users\kjlee>ping 192.168.1.4

Pinging 192.168.1.4 with 32 bytes of data:
Reply from 192.168.1.4: bytes=32 time=1ms TTL=64
Reply from 192.168.1.4: bytes=32 time=1ms TTL=64
Reply from 192.168.1.4: bytes=32 time=1ms TTL=64
Reply from 192.168.1.4: bytes=32 time=1ms TTL=64

Ping statistics for 192.168.1.4:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 1ms, Average = 1ms
```

Figure 9.5. Zephyr Ping Test triggered by Host

9.6. Ping Test (Device to Host)

Zephyr provides a feature that supports network commands for triggering a ping from the device. This section offers guidance on the modifications required in the Zephyr code and the Golden Bootloader C code to enable this feature. Finally, it provides an example command to demonstrate how to trigger a ping from the device.

To ping test from device to host, perform the following steps:

1. To enable Zephyr net command, add the following config in the *lattice_avant_defconfig* file located within Zephyr *project ~/zephyrproject/Zephyr/boards/lattice/avant directory*:

```
60
61  CONFIG_NET_IPV4_HDR_OPTIONS=n
62  CONFIG_NET_SHELL=y
63  CONFIG_NET_SHELL_DYN_CMD_COMPLETION=n
64  CONFIG_NET_TC_TX_COUNT=0
65  CONFIG_NET_TC_RX_COUNT=0
66  CONFIG_ENTROPY_GENERATOR=y
67  CONFIG_TEST_RANDOM_GENERATOR=y
```

Figure 9.6. Enabling Zephyr Net Command

2. Update the post-build script to generate CRC checksum at larger offset (generated image with 512 kB size) in CMakeLists.txt file located in *~/zephyrproject/Zephyr/soc/lattice/avant/* directory as shown in Figure 9.7.

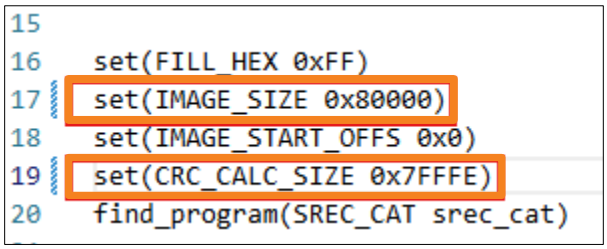


Figure 9.7. Post-Build Script Change to Generate CRC Check Sum at Larger Offset

3. Rebuild the Zephyr code to generate new zephyr_crc.bin file. Refer to the Building the Zephyr section for more details.
4. In the C/C++ bootloader project main.c, update the size of Zephyr image to read from SPI flash (read 512 kB size) as shown in Figure 9.8.

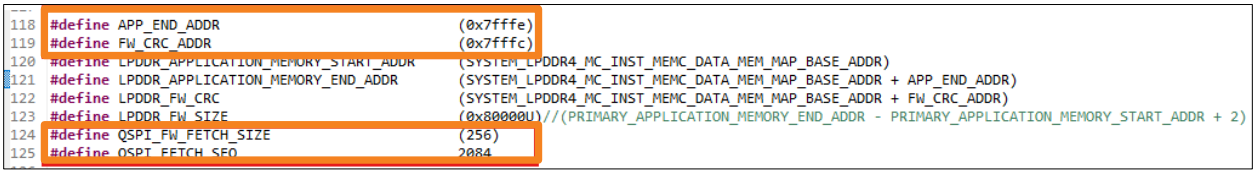


Figure 9.8. C/C++ Golden Bootloader Project Update to Accommodate Larger Zephyr Image Size (512 kB)

5. Build the bootloader project. Then, regenerate the Golden bitstream with newly generated bootloader FW. Refer to steps 7 to 10 in the Building the Bare-metal Bootloader Using the Lattice Propel SDK (Primary and Golden) section on how to build the Golden bootloader FW and regenerate the Golden bitstream using ECO Editor method.
6. Program the updated bitstream and zephyr_crc.bin file to the board. Refer to steps 1 to 20 in the Programming Standalone Golden or Primary GSRD Bitstream and Application Software section for the details.
7. Once the Zephyr boot-up, you can run the net ping commands to execute ping command from device to Host.

   Note: The host firewall setting may block the inbound ping (ICMP) packet from device. Disable the firewall if ping from the device fails).

```
net ping <ping_address>
```

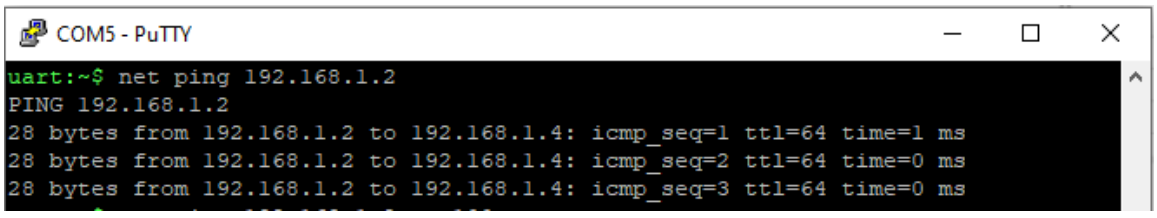


Figure 9.9. Zephyr Ping from Avant

10. Debugging the GSRD

10.1. Debugging Using Reveal Signal

To debug the hardware and software design on the GSRD, the Reveal signals can be added as the trigger to generate waveform to capture the respective signals. Refer to *Chapter 2* and *3* of the Reveal User Guide for Radiant Software document for the steps on how to run the Reveal Inserter and Analyzer.

10.2. Debugging Using the OpenOCD Debugger

Software C/C++ project can be debugged by using the GDB OpenOCD debugger embedded inside the Propel SDK. Refer to the *Programming and On-Chip Debugging Flow* section of the Lattice Propel 2025.1 SDK User Guide (FPGA-UG-02234) for the steps on building and loading the symbol file.

10.3. Debugging with Verbosity Level

Refer to the Appendix B. Enabling Verbosity Level in Software to enable software verbosity level for debugging purposes.

10.4. Error when Building the Software Project

If the error shown in Figure 10.1 appears when building the c_golden_app and c_primary_app software projects, the function *gencmd_quadspi_4kb_erase* in main.c must be replaced with *gencmd_4kb_erase as* shown in Figure 10.2.

```
c:/lscc/propel/2025.2.1/sdk/riscv-none-embed-gcc/bin/../lib/gcc/riscv-none-embed/10.2.0/../../../../riscv-none-embed/bin/ld.exe: ./src/main.o: in function `.L41':
main.c:(.text.octal_spi_diag+0xc0): undefined reference to `gencmd_quadspi_4kb_erase'
collect2.exe: error: ld returned 1 exit status
make[1]: *** [makefile:120: c_golden_app.elf] Error 1
make: *** [makefile:111: all] Error 2
"make all" terminated with exit code 2. Build might be incomplete.
```

Figure 10.1. Lattice Propel SDK Error when Building Software Project

```
227     /* Perform Erase 4kb */
228     //gencmd_quadspi_4kb_erase(octal_spi_c0_inst, flash_addr);
229     gencmd_4kb_erase(octal_spi_c0_inst, flash_addr);
```

Figure 10.2. Function Replacement for 4 kb Erase Command in main.c

Appendix A. Changing the SPIM Settings in the NV Register

Perform the instructions in this appendix if you are using SPI flash that requires 4-Bytes (32-bit) addressing mode.

To change the SPIM settings, perform the following:

1. Perform a one-time step required by JTAG to Program NV Register 1 to modify the default SPI Addressing and Command mode from 24 bits to 32 bits. You need to do this only once for your board. Subsequent programming does not need to program NV Register 1 again.

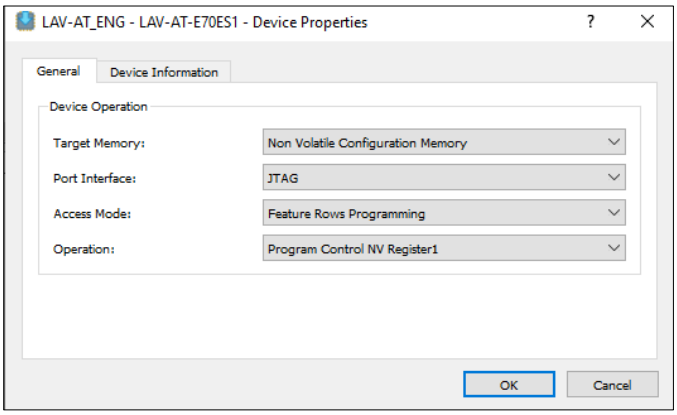


Figure A.1. One-Time Programmable Control NV Register1

2. Click **OK** and click the **Program Device** Icon or go to the menu item, **Run > Program Device**.
3. Change bit 0 to 1 for 32-bit SPIM Address and 32-bit SPIM Commands as shown in Figure A.2. For changing these bits, click once on the 0 values in the Chip Value row.

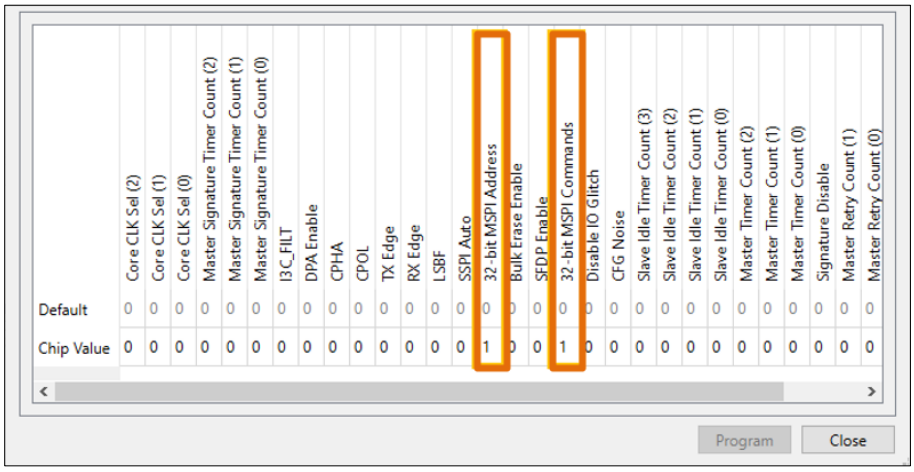


Figure A.2. Settings to Select Chip Value

Note: Update the value of the two highlighted fields. NV Register 1 is an OTP (One-Time Programmable) register.

4. Click **Program**.
5. Power cycle the Evaluation board.

Appendix B. Enabling Verbosity Level in Software

Debuglib is added to implement software verbosity level. There are three verbosities levels: DEBUG_ERROR, DEBUG_INIT, and DEBUG_INFO. Debug Build is enabled by default.

Table B.1. Debuglib Verbose Levels

Debug Level	Description
DEBUG_INIT	Used for all initialization printout. Mandatory. Enabled for both Debug and Release build.
DEBUG_INFO	Used for additional information printout for debugging purposes. Enabled for only Debug build.
DEBUG_ERROR	Used for error printout. Enabled for both Debug and Release build.

To enable verbosity in the software, perform the following:

1. To enable a logging print, you can call DEBUG_MSG (PRINTLEVEL, “Messages”). PRINTLEVEL is the verbosity level selection above. For example, to enable DEBUG_INFO print, you can set the following function:
   ```
   DEBUG_MSG(DEBUG_INFO, “Hello World.”);
   ```
   - For the Debug Build, all three levels of logs are printed.
   - For the Release Build, logging with DEBUG_ERROR and DEBUG_INIT are printed.
2. You can enable the Release Build by right-click on c_primary_app **> Build Configurations > Set Active > Release**.

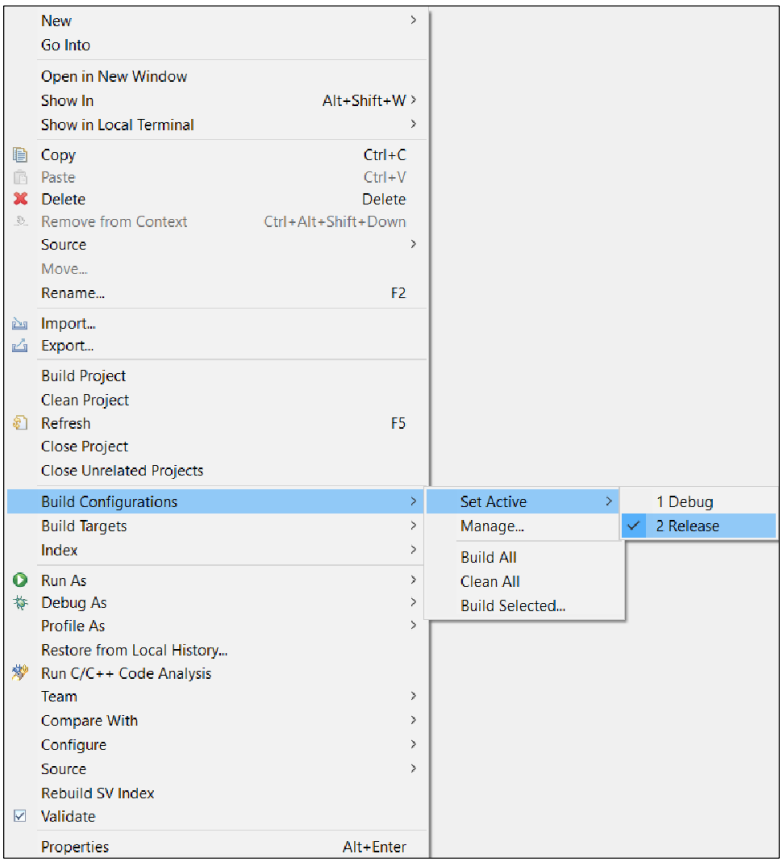


Figure B.1. Set Release Build Configurations

3. Set the defined symbols into the toolchain by right-click on project > **Properties**. In the **C/C++ Build**, select **Settings** > **GNU RISC-V Cross C Compiler** > **Preprocessor** > **Defined symbols (-D)** > Add **LSCC_RELEASE_BUILD** > **Apply and Close**.

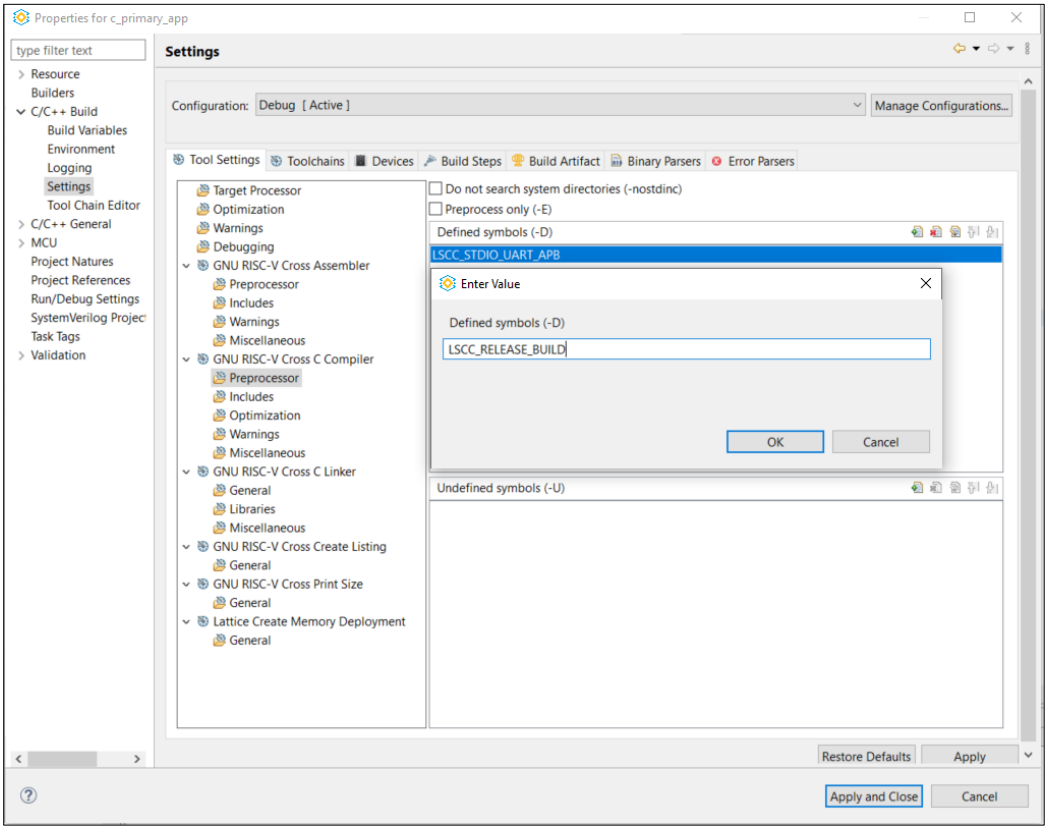


Figure B.2. Add LSCC_RELEASE_BUILD Defined Symbols for Release Build Output

4. Right-click on c_primary_app and click on **Build Project**.

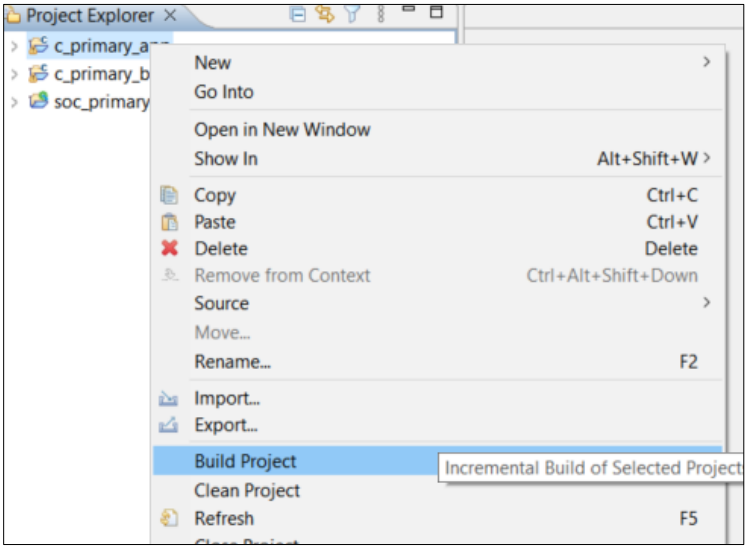


Figure B.3. Build c_primary_app C/C++ Project

These actions enable verbosity levels DEBUG_ERROR and DEBUG_INIT in the Release Build.

Appendix C. Using Different SPI Flash Manufacturer in GSRD Bare-metal Bootloader

There are few SPI flash manufacturers supported by Octal SPI Controller IP and drivers such as Winbond, Macronix, and Micron.

To build the GSRD bootloader for specific SPI flash model, perform the following steps:

1. Launch the Propel SDK and open the c_primary_bootloader or c_golden_bootloader project.
2. Change the flash define parameter in octal_spi_controller.h to the respective SPI flash model (see Figure C.1).

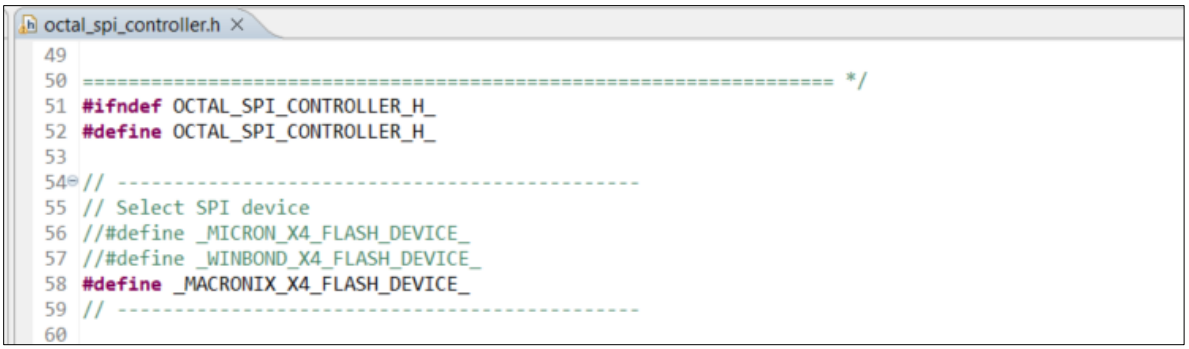


Figure C.1. SPI Flash Manufacturer Changes in octal_spi_controller.h

- _MICRON_X4_FLASH_DEVICE_ = Micron MT25QU128
- _MACRONIX_X4_FLASH_DEVICE_ = Macronix MX25L12833F
- _WINBOND_X4_FLASH_DEVICE_ = Winbond W25Q512JV

3. Redo the steps in the Building the Bare-metal Bootloader Using the Lattice Propel SDK (Primary and Golden) and Building the FreeRTOS Application Software using Lattice Propel SDK (Primary and Golden) section to update the bootloader and FreeRTOS application software.

Appendix D. Using Octal SPI Controller for Read, Write, and Erase Self-Diagnostic Check (FreeRTOS Application Software)

During the software boot-up in FreeRTOS, Octal SPI Controller performs a self-diagnostic check for read, write, and erase operations (enabled by default).

- The Octal SPI Controller erases 4 kB first in the SPI flash at the offset 0x2000000.
- Then, it performs a write of a fixed pattern into the 4 kB region on the same SPI region.
- Finally, it performs a readback to compare if the data is correct from the write.
- If you want to disable the self-diagnostic check, comment out the *octal_spi_diag* function call in FreeRTOS application software main.c.

```
spix8_param_init();

if(spix8_flash_ctl_init(octal_spi_c0_inst, SYSTEM_OSPI_FC_INST_OCTAL_SPI_CONTROLLER_AXI4_MAP_BASE_ADDR,
                        flash_addr_offset))
{
    octal_spi_c0_inst->init_done = SUCCESS;
    DEBUG_MSG(DEBUG_INIT, "Octal SPI Init Done.\r\n");
}
else
    DEBUG_MSG(DEBUG_INIT, "Octal SPI Init Failed\r\n");

gencmd_flash_set_quad_mode(octal_spi_c0_inst, FLASH_QUADSPI_ENABLE);
/* octal spi self-diagnostic check, comment out to skip */
octal_spi_diag(&cmd_buf, &rsp_buf, FLASH_QUADSPI_ENABLE);
```

Figure D.1. Octal SPI Controller Read, Write, and Erase check in FreeRTOS Application Software

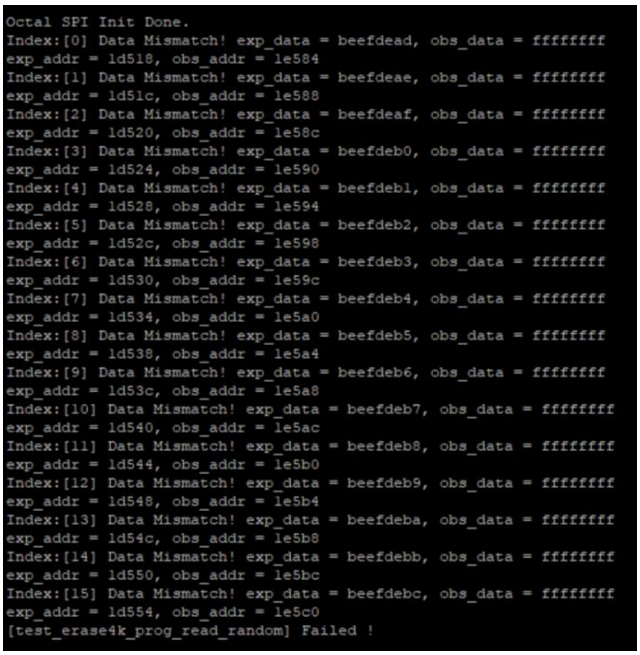


Figure D.2. Self-Diagnostic Check Failed

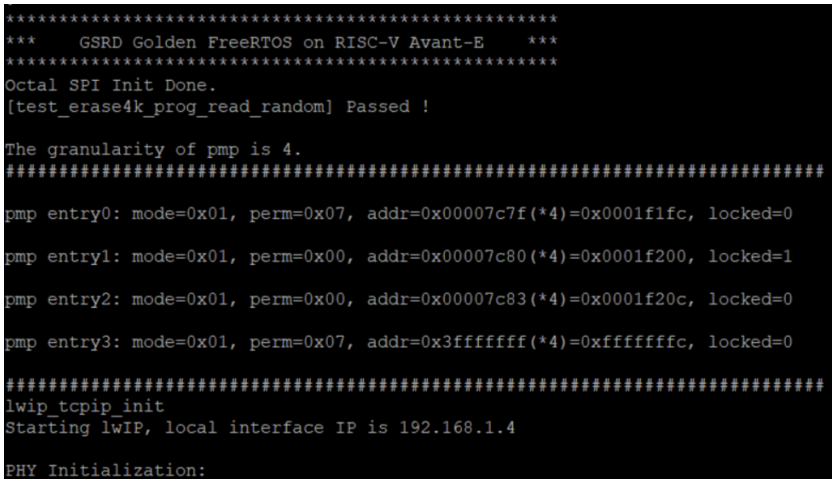


Figure D.3. Self-Diagnostic Check Passed

Appendix E. Configuring SPI Clock (SCK) Pulse Width

To configure the Octal SPI clock output (spi_sck_o) to operate at a frequency different from the Octal SPI clock input (clk_i), the divider register (sck_rate) must be used.

In the Octal SPI Controller IP user interface, ensure that the **Programmable SCK Divider** option is checked (see).

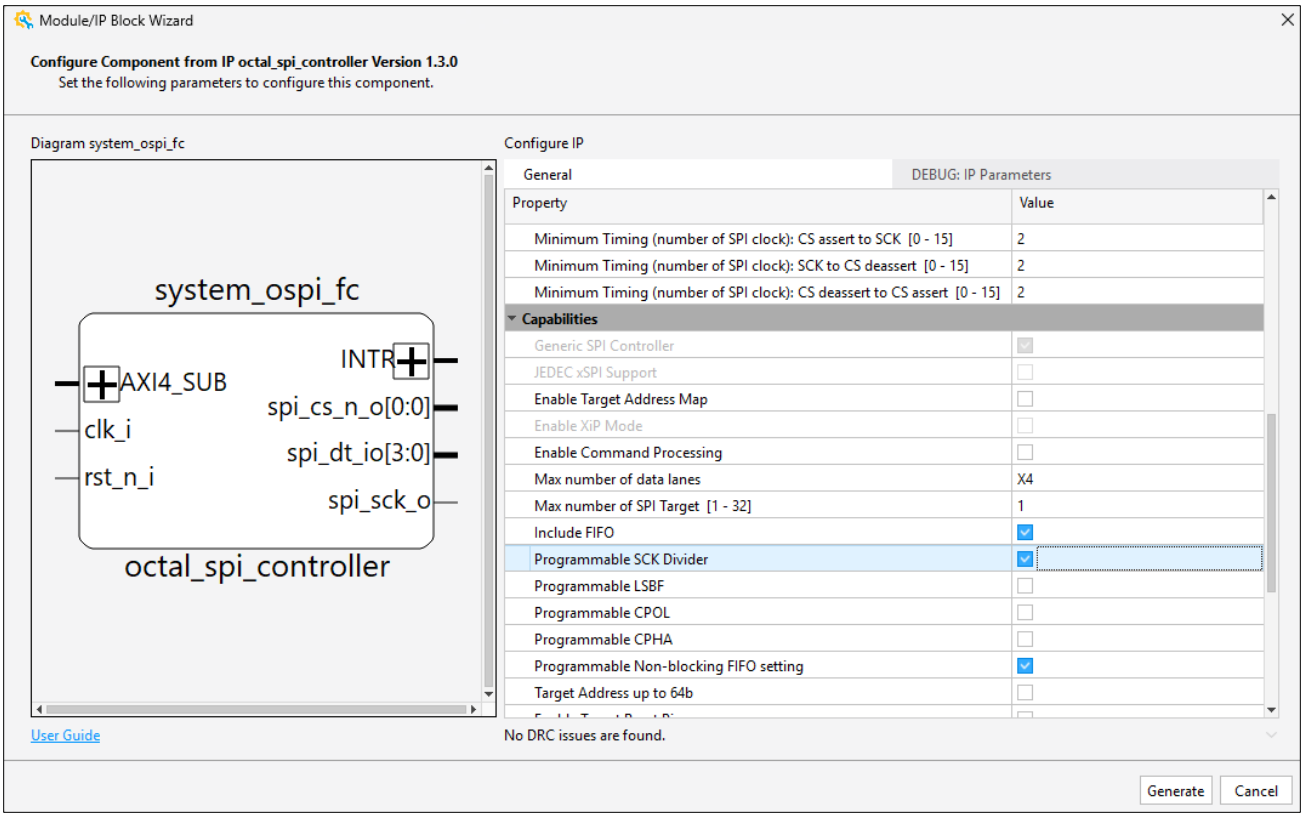


Figure E.1. Programmable SCK Divider Option in the IP User Interface

Refer to the calculations below:

If sck_rate = 0:

spi_sck_o = clk_i

The SPI clock output directly follows the SPI clock input.

If sck_rate ≠ 0:

spi_sck_o = clk_i / (2 × sck_rate)

The SPI clock output is derived by dividing the SPI input clock by twice the value of the divider register.

sck_rate parameter description:

0 – Divide by 1 of clk_i

1 – Divide by 2 of clk_i

2 – Divide by 4 of clk_i

3 – Divide by 6 of clk_i

...

Example:

If the clock source of Octal SPI IP is 200 MHz, the sck_rate must be a value of 2 to get the SPI output clock (sck_sck_o) value of 50 MHz.

Calculation:

SCK Frequency = 200 MHz / (2 × $2^{[1]}$) = 50 MHz

Note: sck_rate value of 2 indicates dividing by 4 (2×2) in the calculation.

The sck_rate configuration can be found in the spix8_param_init function located in the bootloader's main.c (see below).

```
void spix8_param_init(void){

    unsigned int sck_rate;
    sck_rate = 1;

    octal_spi_c0.init_done                  = FAILURE;
    octal_spi_c0.base_addr                  = SYSTEM_OSPI_FC_INST_OCTAL_SPI_CONTROLLER_AXI4_MAP_BASE_ADDR;
    octal_spi_c0.max_num_lane               = SYSTEM_OSPI_FC_INST_MAX_NUMLANE;
    octal_spi_c0.sys_clk_freq               = SYSTEM_OSPI_FC_INST_CLKI_FREQ;
    octal_spi_c0.spi_io_width               = SPIX8_IO_X1;
```

Figure E.2. Octal SPI Controller sck_rate Configuration

References

- Avant-E Evaluation Board User Guide (FPGA-EB-02057)
- APB Interconnect IP User Guide (FPGA-IPUG-02054)
- AXI4 Interconnect IP User Guide (FPGA-IPUG-02196)
- AXI4 to APB Bridge IP User Guide (FPGA-IPUG-02198)
- GPIO IP Core User Guide (FPGA-IPUG-02076)
- Lattice IP Packager 2025.1 (FPGA-UG-02236)
- Lattice Propel 2025.1.1 Builder User Guide (FPGA-UG-02298)
- Lattice Propel 2025.1 SDK User Guide (FPGA-UG-02234)
- Lattice Radiant Timing Constraints Methodology (FPGA-AN-02059)
- DDR Memory Controller IP Core (FPGA-IPUG-02208)
- Octal SPI Controller IP Core User Guide (FPGA-IPUG-02248)
- RISC-V RX CPU IP User Guide (FPGA-IPUG-02298)
- SGDMA Controller IP Core User Guide (FPGA-IPUG-02131)
- System Memory IP User Guide (FPGA-IPUG-02073)
- Tri-Speed Ethernet IP Core User Guide (FPGA-IPUG-02084)
- UART IP Core User Guide (FPGA-IPUG-02105)
- Lattice Radiant Software 2025.1 User Guide
- Reveal User Guide for Radiant Software
- Lattice Propel Design Environment web page
- Lattice Radiant Software web page
- Lattice Solutions IP Cores 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 1.2, July 2026

Section	Change Summary
All	Made editorial fixes across the document.
Customizing the GSRD	Added IP Version Refresh without Structural Modification section.
Debugging the GSRD	Added Error when Building the Software Project section.

Revision 1.1, December 2025

Section	Change Summary
Functional Description	Made editorial fix to change KB to *kB* in Table 2.6. Address Map of GHRD.
Building the RISC-V Zephyr	Added this section to include information on configuring, building, and running Zephyr in the GSRD.
Appendix E. Configuring SPI Clock (SCK) Pulse Width	Made editorial fix to remove number from the *Note* text.

Revision 1.0, October 2025

Section	Change Summary
All	Initial release.

www.latticesemi.com