Automate Stack 3.1

Reference Design

FPGA-RD-02284-1.0

April 2024

Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults, and all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE 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

Contents....3
Acronyms in This Document....10
1. Introduction....11
1.1. Components....12
2. Design Overview....13
2.1. Theory of Operation....13
2.2. FPGA Design....14
2.2.1. Main System....14
2.2.2. Lattice Main System 3.1 Architecture....17
2.2.3. Node System....20
2.3. EtherConnect IP Design Details....22
2.3.1. Overview....22
2.3.2. Architecture....24
2.3.3. Register Map....24
2.4. FIFO DMA....25
2.5. UDP Stack....26
2.6. LPDDR4 Controller....28
2.7. QSPI Flash controller....29
2.8. Multi-Port Memory Controller IP Design Details....30
2.9. Scatter Gather DMA IP Design Details....33
2.10. CNN Coprocessor Unit (CCU)....34
2.11. Motor Control and PDM Data Collector....37
2.12. SPI Manager IP Design Details....44
2.12.1. Overview....45
2.12.2. SPI Manager Register Map....45
2.12.3. Programming Flow....46
2.13. I^2C Manager IP Design Details....47
2.13.1. Overview....47
2.13.2. I^2C Manager Register Map....48
2.13.3. Programming Flow....48
2.14. UART IP Design....50
2.14.1. Overview....50
2.14.2. Programming Flow....51
2.15. TSE MAC....53
2.16. SGMII IP Design....54
2.17. FPGA Config Module Design....54
2.18. SFP Config Design Details....56
3. Resource Utilization....57
4. Software APIs....58
4.1. Main System APIs....58
4.1.1. Tasks of the Main System....58
4.1.2. OPCUA PubSub :....60
4.1.3. Create_UADP_NetworkMessage....60
4.1.4. GroupHeader....61
4.1.5. Extended NetworkMessage Header....62
4.2. Node System APIs....64
4.2.1. Tasks of the Node System....64
4.2.2. API Calls....64
5. Communications....67
5.1. Communication between Host and Main System....67
5.1.1. Messages from Host to Main System....67
5.1.2. Messages from Main System to Host....67

5.2. Communication between Main System and Node System(s) 67
5.2.1. Messages from Main System to Node System 67
5.2.2. Messages from Node System to Main System 67
Appendix A. Predictive Maintenance with TensorFlow Lite 68
A.1. Setting Up the Linux Environment for Neural Network Training 68
A.1.1. Installing the NVIDIA CUDA and cuDNN Library for Machine Learning Training on GPU 68
A.1.2. Setting Up the Environment for Training and Model Freezing Scripts 70
A.1.3. Installing the TensorFlow Version 1.15 71
A.1.4. Installing the Python Package 72
A.2. Creating the TensorFlow Lite Conversion Environment 73
A.3. Preparing the Dataset 73
A.3.1. Dataset Information 74
A.4. Preparing the Training Code 74
A.4.1. Training Code Structure 74
A.4.2. Generating tfrecords from Augmented Dataset 75
A.4.3 Neural Network Architecture 75
A.4.4. Training Code Overview 77
A.4.5. Training from Scratch and/or Transfer Learning 84
A.5. Creating Frozen File 87
A.5.1. Generating .pbtxt File for Inference 87
A.5.2. Generating the Frozen (.pb) File 87
A.6. TensorFlow Lite Conversion and Evaluation 88
A.6.1. Converting Frozen Model to TensorFlow Lite 88
A.6.2. Evaluating TensorFlow Lite Model 88
A.6.3. Converting TensorFlow Lite To C-Array 89
Appendix B. Setting Up the Wireshark Tool 90
Appendix C. Generating Automate Stack 3.1 Propel Patch and Bitstream 92
C.1. Installing the Propel SDK 2023.2 92
C.2. Generating the Binary 97
C.3.1. Primary Main System 97
C.3.2. Golden Main System 106
C.3.3. Node System 112
C.4. Generating the Bit File 117
C.4.1. Primary Main System 117
C.4.2. Golden Main System 122
C.4.3. Node System 128
Appendix D. Creating the MCS File 134
References 137
Technical Support Assistance 138
Revision History 139

Figures

Figure 1.1. Top Level Block Diagram of Automate Stack 3.1 12
Figure 2.1. Lattice Automate Stack 3.1 Top Level Block Diagram 13
Figure 2.2. GSRD Architecture 15
Figure 2.3. Deployment Tool Multi-Boot Tab 16
Figure 2.4. Main System Architecture 18
Figure 2.5. Client to Server Data Flow 19
Figure 2.6. Node System Architecture 21
Figure 2.7. Packet Structure 23
Figure 2.8. UDP Stack Top Level Architecture 27
Figure 2.9. UDP Stack IPV4 address configuration 28
Figure 2.10. Memory Controller IP Core Functional Diagram 29
Figure 2.11. QSPI Flash Controller Block Diagram 30
Figure 2.12. MPMC Top-level Block Diagram 31
Figure 2.13. SGDMA IP Functional Diagram 33
Figure 2.14. Motor Controller Interface with Motor 37
Figure 2.15. SPI Manager IP Core Block Diagram 45
Figure 2.16. I^2C Manager Controller IP Core Functional Diagram 47
Figure 2.17. UART IP Core Functional Block Diagram 50
Figure 2.18. UART Data Format 53
Figure 2.19. Classic TSEMAC IP Top-Level Block Diagram 53
Figure 2.20. SGMII IP Settings 54
Figure 2.21. CONFIG_LMMIA Primitive and OSC Primitive Connection 55
Figure 2.22. LMMI LSC_REFRESH Command Execution 55
Figure 2.23. SFP Config User Interface 56
Figure 4.1. UADP Version 61
Figure 4.2. UADP Message Packet Header 61
Figure 4.3. Create_UADP_NetworkMessage 62
Figure 4.4. UADP Network Message Format 63
Figure A.1. Download CUDA Repo 68
Figure A.2. Install CUDA Repo 68
Figure A.3. Fetch Keys 68
Figure A.4. Update Ubuntu Packages Repositories 69
Figure A.5. CUDA Installation 69
Figure A.6. cuDNN Library Installation 69
Figure A.7. Anaconda Installation 70
Figure A.8. Accept License Terms 70
Figure A.9. Confirm/Edit Installation Location 70
Figure A.10. Launch/Initialize Anaconda Environment on Installation Completion 70
Figure A.11. Anaconda Environment Activation 71
Figure A.12. TensorFlow Installation 71
Figure A.13. TensorFlow Installation Confirmation 71
Figure A.14. TensorFlow Installation Completion 71
Figure A.15. Easydict Installation 72
Figure A.16. Joblib Installation 72
Figure A.17. Keras Installation 72
Figure A.18. OpenCV Installation 72
Figure A.19. Pillow Installation 73
Figure A.20. Predictive Maintenance Dataset 74
Figure A.21. Training Code Directory Structure 74
Figure A.22. Training Code Flow Diagram 77
Figure A.23. Code Snippet: Hyper Parameters 78
Figure A.24. Code Snippet: Build Input 78

Figure A.25. Code Snippet: Parse tfrecords 79
Figure A.26. Code Snippet: Convert Image to Grayscale 79
Figure A.27. Code Snippet: Convert Image to BGR and Scale the Image 79
Figure A.28. Code Snippet: Create Queue 79
Figure A.29. Code Snippet: Add Queue Runners 80
Figure A.30. Code Snippet: Create Model 80
Figure A.31. Code Snippet: Fire Layer 80
Figure A.32. Code Snippet: Convolution Block 81
Figure A.33. Code Snippet: Feature Depth Array for Fire Layers 81
Figure A.34. Code Snippet: Forward Graph Fire Layers 82
Figure A.35. Code Snippet: Loss Function 82
Figure A.36. Code Snippet: Optimizers 82
Figure A.37. Code Snippet: Restore Checkpoints 83
Figure A.38. Code Snippet: Save .pbtxt 83
Figure A.39. Code Snippet: Training Loop 83
Figure A.40. Code Snippet: _ LearningRateSetterHook 83
Figure A.41. Code Snippet: Save Summary for Tensorboard 84
Figure A.42. Code Snippet: logging hook 84
Figure A.43. Predictive Maintenance – Run Script 84
Figure A.44. Predictive Maintenance – Trigger Training 85
Figure A.45. Predictive Maintenance – Trigger Training with Transfer Learning 85
Figure A.46. Predictive Maintenance – Training Logs 85
Figure A.47. Predictive Maintenance – Confusion Matrix 85
Figure A.48. Tensorboard – Launch 86
Figure A.49. Tensorboard – Link Default Output in Browser 86
Figure A.50. Checkpoint Storage Directory Structure 86
Figure A.51. Generated '.pbtxt' for Inference 87
Figure A.0.52. Run genpb.py to Generate Inference .pb 87
Figure A.53. Frozen Inference .pb Output 88
Figure B.1. Wireshark Downloadable Link 90
Figure B.2. Wireshark Tool – Ethernet selection 90
Figure B.3. Wireshark Tool – Write udp.port == 1486 90
Figure B.4. Source and Destination UDP Packet 91
Figure B.5. Wireshark Tool – First UDP Packet 91
Figure C.1. Propel Application 92
Figure C.2. Allow Permission 92
Figure C.3. Lattice Propel 2023. 2 Installation Wizard 93
Figure C.4. Select Installation Folder 93
Figure C.5. Install Components 94
Figure C.6. Accept the License 94
Figure C.7. Start Menu Shortcut 95
Figure C.8. Install the Propel SDK Application 95
Figure C.9. Installation Process 96
Figure C.10. Installation Complete 96
Figure C.11. License Path 97
Figure C.12. Automate 3.1 Propel Patch 97
Figure C.13. Propel 2023.2 Application 97
Figure C.14. Select Directory 97
Figure C.15. Import Project 98
Figure C.16. Existing Project into Workspace 98
Figure C.17. Import Project 99
Figure C.18. Properties 100
Figure C.19. C/C++ build settings 101
Figure C.20. Manage Configuration – Release: Set Active 101

Figure C.21. Manage Configuration: Apply and Close 102
Figure C.22. Clean project Configurations 103
Figure C.23. Console 103
Figure C.24. Build Project 104
Figure C.25. Completing Process 104
Figure C.26. Clean All Configurations 105
Figure C.27. Console 105
Figure C.28. Build All Configurations 106
Figure C.29. Completing Process 106
Figure C.30. Propel 2023.2 Application 106
Figure C.31. Select Directory 107
Figure C.32. Import Project 107
Figure C.33. Existing Project into Workspace 107
Figure C.34. Import Project 108
Figure C.35. Clean Project Configurations 109
Figure C.36. Console 109
Figure C.37. Build Project 110
Figure C.38. Completing Process 110
Figure C.39. Clean All Configurations 111
Figure C.40. Console 111
Figure C.41. Build All Configurations 112
Figure C.42. Completing Process 112
Figure C.43. Propel Application 112
Figure C.44. Select Directory 113
Figure C.45. Import Project 113
Figure C.46. Existing Project into Workspace 113
Figure C.47. Select Project 114
Figure C.48. Clean All 115
Figure C.49. Console 115
Figure C.50. Build All 116
Figure C.51. Completing Process 116
Figure C.52. soc_main_system.sbx 117
Figure C.53. System Initialization File 117
Figure C.54. Validate Button 118
Figure C.55. Generate SGE Button 118
Figure C.56. Radiant Tool Button 118
Figure C.57. soc_main_system.rdf file 118
Figure C.58. LFCPNX-100-9LFG672I 118
Figure C.59. Lattice Radiant Device Selector for Main System 119
Figure C.60. Strategy for Build Generation for Main System 119
Figure C.61. MAP Analysis Setting for Main System Bit File Generation 120
Figure C.62. PAR Setting for Main System Bit File Generation 120
Figure C.63. PAR Timing Analysis Setting for Main System Bitfile Generation 121
Figure C.64. IP Evaluation 121
Figure C.65. Run All Button 122
Figure C.66. Bitstream File 122
Figure C.67. soc_main_system.sbx 122
Figure C.68. System Initialization File 123
Figure C.69. Validate Button 123
Figure C.70. Generate SGE Button 123
Figure C.71. Radiant Tool Button 124
Figure C.72. soc_main_sysyem.rdf File 124
Figure C.73. LFCPNX-100-9LFG672I 124
Figure C.74. Lattice Radiant Device Selector for Main System 125

Figure C.75. Strategy for Build Generation for Main System 125
Figure C.76. MAP Analysis Setting for Main System Bit File Generation 126
Figure C.77. PAR Setting for Main System Bit File Generation 126
Figure C.78. PAR Timing Analysis Setting for Main System Bitfile Generation 127
Figure C.79. IP Evaluation 127
Figure C.80. Run All 128
Figure C.81. Bitstream File 128
Figure C.82. soc_node.sbx 128
Figure C.83. System0 Initialization 129
Figure C.84. Validate Button 129
Figure C.85. Generate SGE Button 129
Figure C.86. Radiant Tool Button 129
Figure C.87. soc_node.rdf file 130
Figure C.88. LFD2NX-40-8BG256C 130
Figure C.89. Lattice Radiant Device Selector for Node System 130
Figure C.90. Strategy for Build Generation for Node System 131
Figure C.91. MAP Analysis Setting for Node System Bit File Generation 131
Figure C.92. PAR Setting for Node system Bit File Generation 132
Figure C.93. PAR Timing Analysis Setting for Node System Bit File Generation 132
Figure C.94. IP Evaluation 133
Figure C.95. Run All Button 133
Figure C.96. Bitstream File 133
Figure D.1. Deployment tool 134
Figure D.2. Creating New Deployment for Multi-Boot 134
Figure D.3. Select Input File Window 134
Figure D.4. Advanced SPI Flash Options - Multi-Boot Tab Window 135
Figure D.5. Select Output File Window 135
Figure D.6. Generate Deployment Window 136

Tables

Table 2.1. GSRD Memory Map 17
Table 2.2. Main System Memory Map 20
Table 2.3. Memory Map of Node System 21
Table 2.4. EtherConnect IP Global Registers 24
Table 2.5. EtherConnect IP Chain 1 Registers 24
Table 2.6. FIFO DMA Register Map 25
Table 2.7. FIFO DMA Control Registers 25
Table 2.8. DEST_BASE_ADDR Register 25
Table 2.9. DEST_END_ADDR Register 26
Table 2.10. Write Status Register 26
Table 2.11. Read Status Register 26
Table 2.12. MPMC Register Map 31
Table 2.13. Register Map of SGDMA IP 34
Table 2.14. CNN Coprocessor Unit Registers 34
Table 2.15. CNN Coprocessor unit control register 35
Table 2.16. CNN Coprocessor Unit Register 35
Table 2.17. Sign Select Configuration Register 35
Table 2.18. Input Offset Configuration Register 35
Table 2.19. Filter Offset Configuration Register 35
Table 2.20. Filter Offset Configuration Register 36
Table 2.21. Input Depth Configuration Register 36
Table 2.22. Input Data Address Configuration Register 36
Table 2.23. Filter Data Address Configuration Register 36
Table 2.24. CNN Coprocessor Unit Output Register 36
Table 2.25. Predictive Maintenance and Motor Control Registers 37
Table 2.26. Motor Control 0 – Minimum RPM 38
Table 2.27. Motor Control 1 – Maximum RPM 38
Table 2.28. Motor Control 2 – RPM PI Control Loop Integrator Gain (kI) 38
Table 2.29. Motor Control 3 – RPM PI Control Loop Proportional Gain (kP) 38
Table 2.30. Motor Control 4 – Torque PI Control Loop Integrator Gain (kI) 39
Table 2.31. Motor Control 5 – Torque PI Control Loop Proportional Gain (kP) 39
Table 2.32. Motor Control 6 – Synchronization Delay and Control 39
Table 2.33. Motor Control Register 7 – Target RPM 40
Table 2.34. Motor Control Register 8 – Target Location 40
Table 2.35. Motor Control Register 9 – Current Location 40
Table 2.36. Motor Status Register 0 – RPM 40
Table 2.37. Motor Status Register 1 41
Table 2.38. Predictive Maintenance Control Register 0 41
Table 2.39. Predictive Maintenance Control Register 1 42
Table 2.40. Predictive Maintenance Status Register 42
Table 2.41. Predictive Maintenance Current/Voltage Data Register 43
Table 2.42. Predictive Maintenance Current/Voltage Data Register 43
Table 2.43. Versa Board Switch Status Register 43
Table 2.44. Versa Board LED and PMOD Control Register 44
Table 2.45. SPI Manager Register Map 45
Table 2.46. I^2C Manager IP Core Registers Summary 48
Table 2.47. UART Register Map 51
Table 3.1. Main System Resource Utilization 57
Table 3.2. Node System Resource Utilization 57
Table A.1. Predictive Maintenance Training Network Topology 75

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
AHBL	Advanced High-performance Bus-Lite
AI	Artificial Intelligence
API	Application Programming Interface
BLDC	Brushless DC
CCU	CNN Coprocessor Unit
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access
FIFO	First-In-First-Out
ISR	Interrupt Service Routines
LPDDR4	Low Power Double Data Rate Generation 4
ML	Machine Learning
QSPI	Quad Serial Peripheral Interface
RISC-V	Reduced Instruction Set Computer-V
RTL	Register-Transfer Level
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Data gram Protocol
TSEMAC	Tri-Speed Ethernet Media Access Controller

1. Introduction

Lattice Automate™ Stack provides a solution for industrial automation that includes predictive maintenance using ML/AI, communication over Ethernet cable and a BLDC motor control IP implemented in RTL. The solution enables user to control multiple motors connected to node systems that are chained using Ethernet cable. The main system that synchronizes operations of node system also runs neural network trained using RISC-V and CNN Coprocessor for predictive maintenance. The entire solution can work with or without external host. The reference design is provided with a user interface that runs on host and controls motor operations. The user interface also displays the status of motor and alerts user when motor requires maintenance. User can use all APIs provided with this reference design and can implement entire system without host system. In this case, the C/C++ code running on RISC-V sends required commands to control motors. The entire system with all sub-components is shown in further sections.

Lattice Automate Stack 1.0 supports web-based user interface which is running on host (system PC) and single chain of nodes for controlling the motors.

Lattice Automate Stack 1.1 supports two chains of nodes which can be connected to one main system board. All nodes are synchronized physically. Main system supports dynamic pulse-based system synchronization scheme, in which it checks nodes disconnection during runtime and compensate clock ppm to calculate synchronization delay. It supports OPC UA server/client-based user interface, which is running on host PC and client are running on Raspberry Pi board.

Lattice Automate Stack 2.0 supports all features of Lattice Automate Stack 1.1. It supports MQTT broker/client-based host application, Python Interface as host control, and supports PCIe® interface as host for high-speed applications. In the node side, it has motor IP for motor-based features and standard SPI Manager and I^2C Manager interfaces to connect various peripherals (sensors) into system.

Lattice Automate Stack 3.0 supports free RTOS (RISC-V) CPU IP and OPC-UA client-based host PC which is connected to CertusPro™-NX (Main system) using Ethernet cable. Host PC and Main system can also be connected to a common ethernet switch. OPC-UA server is running on free RTOS (RISC-V) in main board and OPC-UA client is running on host PC. Communication between Host PC(Client) and free RTOS (server) is established over 1G ethernet network. SGMII, TSE MAC, UDP Stack and LPDDR4 and Multiport Extension IPs are used to enable data exchange between RISC-V and Host PC. AHBL bus interface is replaced by AXI4 bus interface. IPs with AXI4 Manager communicates using common AXI4 Interconnect with other AXI4 subordinate based IPs connected interconnect. AXI4 bus interface has more throughput than AHBL and allows the CPU to run on higher frequency as well (up to 100 MHz). It also allows parallel data transfer between subordinate and manager. This main system SOC supports only 1G port for node system chain connection due to 1G port and resources limitations on the target board. Figure 1.1 shows the Automate Stack solution and its subcomponents.

In Lattice Automate Stack 3.1, the Golden System reference design is used. The Golden System reference design can detect the integrity of binaries and execute appropriate program. The system has two FPGA images: Primary Image and Golden Image. If Primary Image integrity check fails, the system switches to Golden Image. Automate Stack 3.0 functionalities are ported on GSRD based SOC. Automate Stack 3.1 supports OPC-UA based packet exchange b/w Main and Node system for various data transfers.

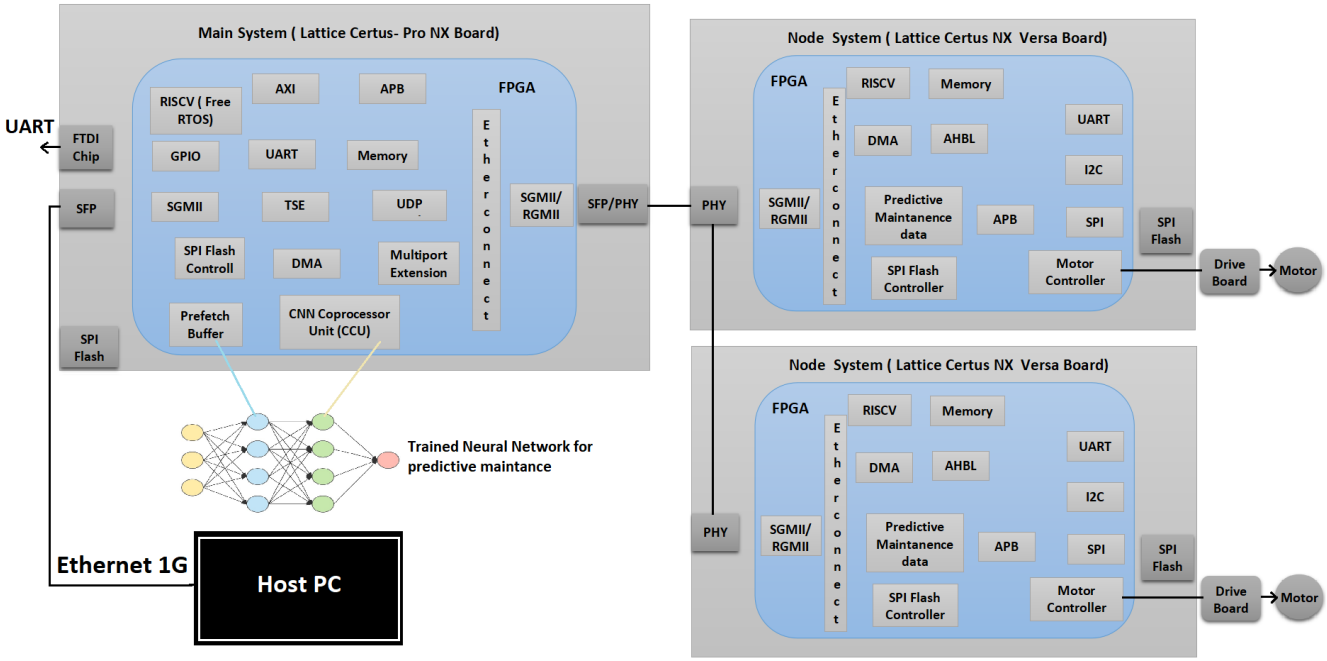

Figure 1.1. Top Level Block Diagram of Automate Stack 3.1

1.1. Components

The Automate Stack 3.1 release includes the following components:

- System on Chip (SOC)
  - Main System IPs
    - EtherConnect IP (With SGMII/RGMII (PHY or SFP), FIFO DMA, CNN Coprocessor Unit (CCU), SPI Flash Controller, Multiport extension, UDP Stack, SGMII TSE MAC, and Reset Synchronizer.
  - Node System IPs
    - EtherConnect IP (With SGMII/RGMII (PHY or SFP), FIFO DMA, BLDC motor control IP, and Data collector for predictive maintenance
    - Modbus, I^2C Manager, and SPI Manager
- Software
  - Firmware (APIs)
    - APIs to send instructions to motor control IP, collect status of motors and collect data for predictive maintenance Compiled TensorFlow-Lite C++ library for RISC-V (Required for neural network inference).
  - User Interface
    - Controls motor, collects status and data for predictive maintenance, displays warning when maintenance required.
- Machine Learning
  - Trained Neural Network for predictive maintenance.
  - Script to train network with user collected data.

Note: The generic RISC-V subsystem components are excluded from the list of components.

2. Design Overview

2.1. Theory of Operation

There are two different SoCs to be released:

- GSRD – System Hardware (SoC) supports RISC-V RX, MPMC, DM, and QSPI controller over AXI interface and comprises of Boot loader, Primary image, and Golden image.
- Lattice Automate Stack 3.1 – Automate Stack 3.0 Functionality ported on GSRD-based SoC.

Figure 2.1 shows the overall architecture of the Automate demo system. The Automate Stack 3.1 consists of one Main System (MS) and multiple Node Systems (NS), maximum of eight in a chain. The host is connected to the MS through ethernet cable. The application software, with the user interface running on host, can send commands to the MS and receive motor maintenance data from the system for AI training. The MS can propagate the commands to NS using OP-CUA packets for motor control and gather maintenance data from NS.

Hosts can also send/receive data from different peripherals connected to node other than motor.

For main system, LFCPNX-100-9LFG672C/I device is used for the demo design. For node system, the Certus™-NX Versa Board is used for the demo design.

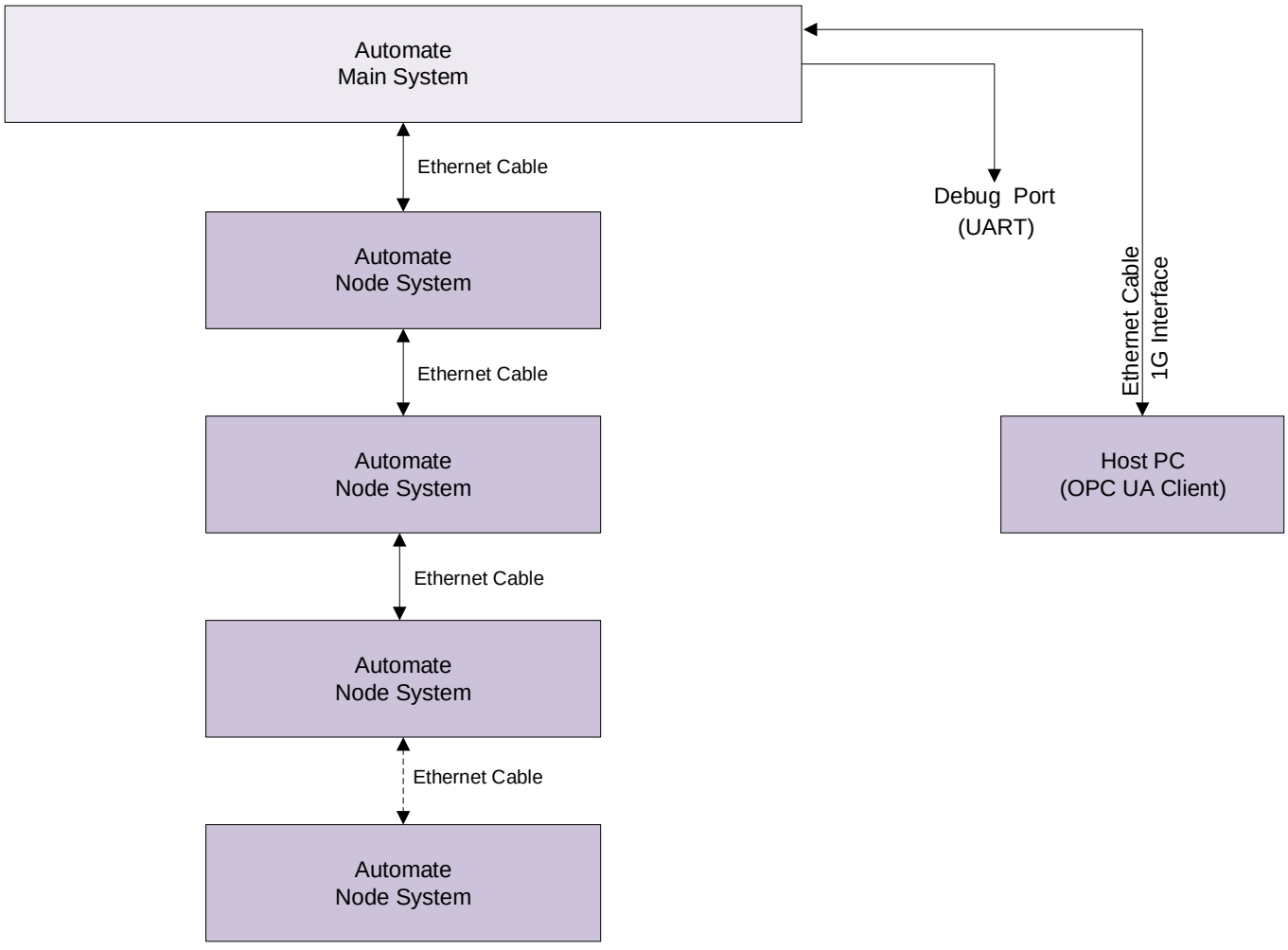

Figure 2.1. Lattice Automate Stack 3.1 Top Level Block Diagram

2.2. FPGA Design

2.2.1. Main System

2.2.1.1. GSRD Architecture

Figure 2.2 shows the GSRD Architecture which has two AXI4 Interconnects. Interconnect-1 has three managers and five subordinates:

- Three Managers:
  - RISC-V RX CPU Instruction Port
  - RISC-V RX CPU Data Port
  - SGDMA connected through Interconnect-2
- Five Subordinates
  - System Memory
  - AXI2APB Bridge
  - MPMC
  - SGDMA
  - SPI Flash Controller

The RISC-V RX CPU and AXI Interconnect-2 (SGDMA) can access data to the shared memory Data RAM, MPMC, SPI Flash Controller, and AXI2APB bridge directly and UART, TSE MAC, Memory Controller, SGDMA, FPGA Config module, and GPIO through AXI2APB bridge. UART and GPIO can generate interrupts to RISC-V CPU.

The Interconnect 2 has one manager and two subordinates:

- One Manager – SGDMA
- Two Subordinates – MPMC and Interconnect-1

The SGDMA can access data to the MPMC and AXI Interconnect-1.

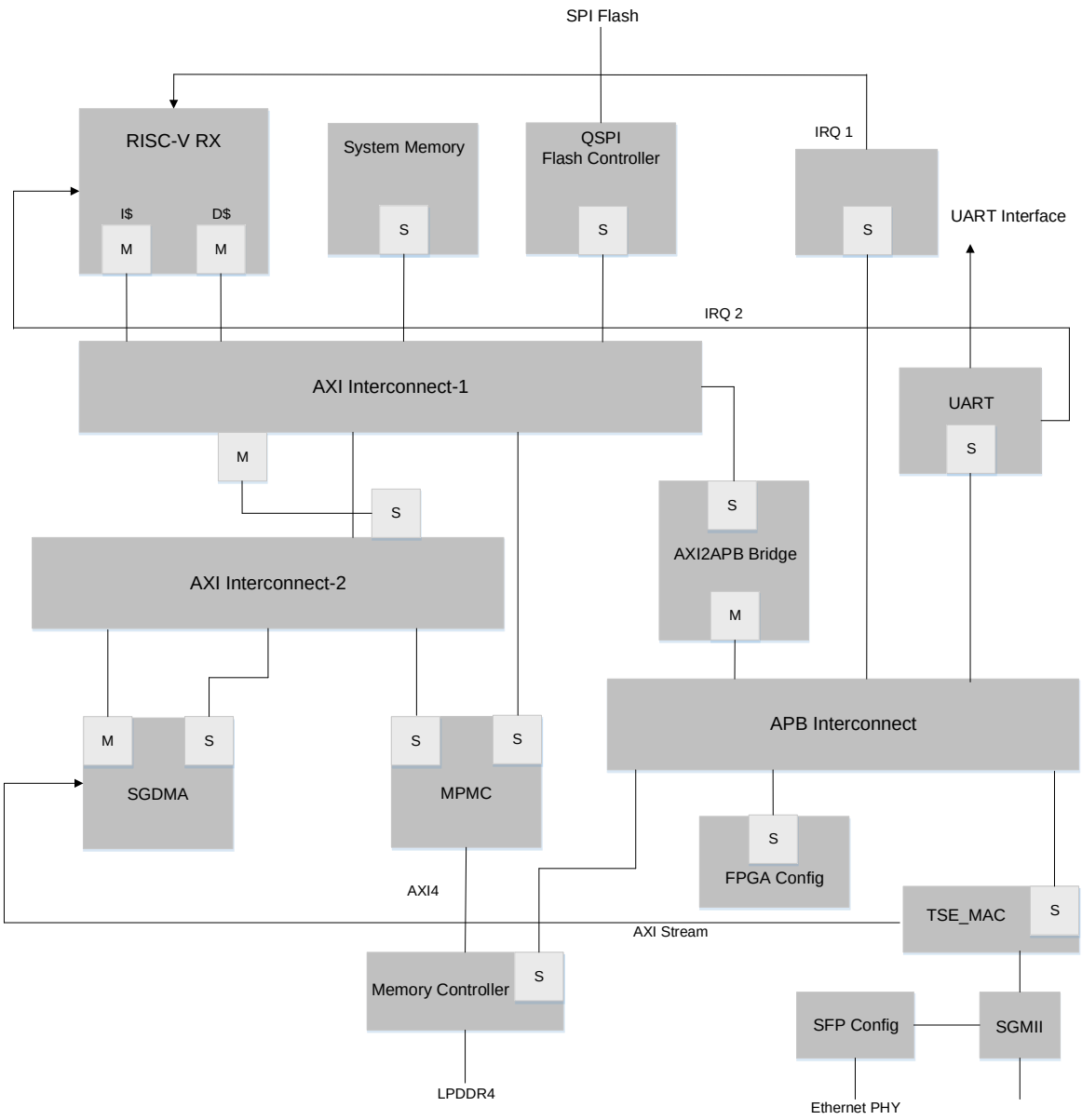

Figure 2.2. GSRD Architecture

2.2.1.2. GSRD Data Flow

Multiboot Flow

The Certus Pro-NX device multi-boot supports booting from up to six patterns that reside in an external SPI Flash device. The patterns include a primary pattern, a golden pattern, and up to four alternate patterns, designated as alternate pattern 1 to alternate pattern 4. The CertusPro-NX device boots by loading the primary pattern from the internal or external Flash. If loading of the primary pattern fails, the CertusPro-NX device attempts to load the Golden pattern. When reprogramming of the bitstream is triggered through the toggling of the PROGRAMN pin or receiving a REFRESH command, the alternate pattern 1 is loaded. Subsequent PROGRAMN/REFRESH event loads the next pattern defined in the multi-boot configuration. The bitstream pattern sequence, target address of the Golden pattern, and target addresses of the alternate patterns are defined during the multi-boot configuration process in the Lattice Radiant Deployment Tool as shown in Figure 2.3.

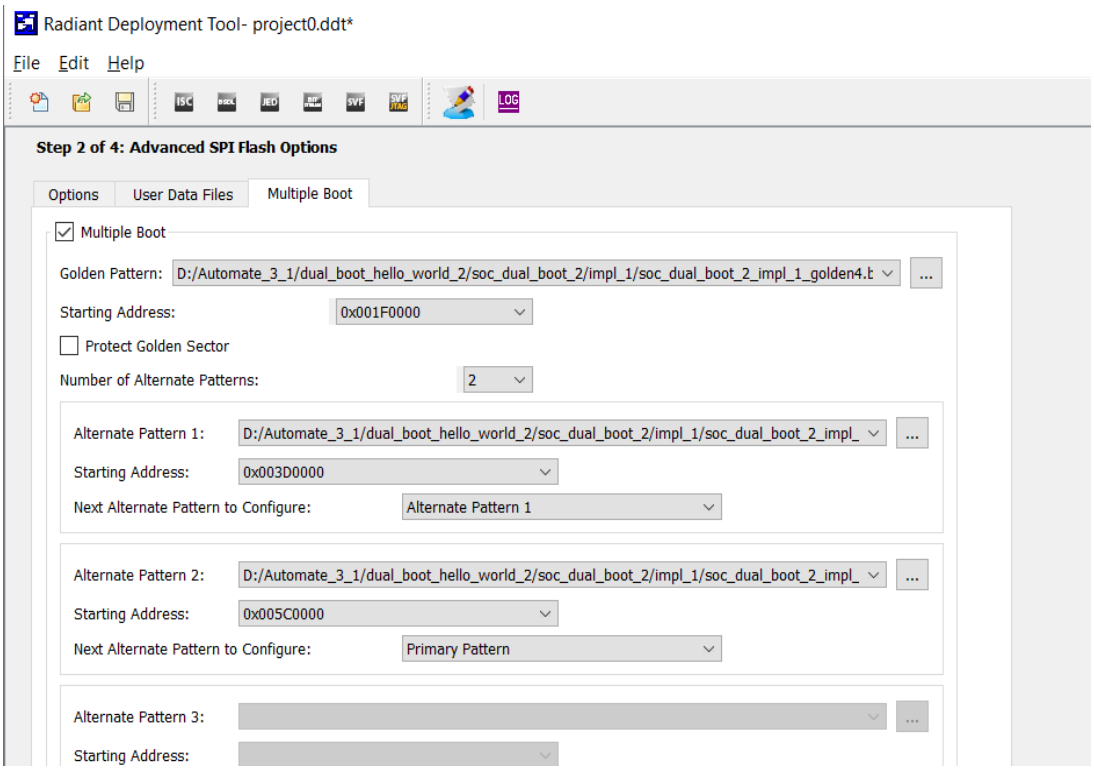

Figure 2.3. Deployment Tool Multi-Boot Tab

2.2.1.3. GSRD Flow

The GSRD design has two firmware binaries and two FPGA bit files. One set of binary and bit file is golden and the other one is primary. The golden image works as a baseline version of the system. The primary image is an updated version of the system.

The boot loader firmware supports the CRC checking and switching between the primary Image and golden image. The firmware has the option to manually boot the FPGA image-based on CRC check.
Upon performing a CRC check on the binary file, if the primary binary is corrupted, the booting occurs from the golden one but the bit file must also switch to golden. There is a firmware code in flash to switch the bit file to golden and the same happens when the primary bit file is corrupted.

The booting is done from one of the two sets of binary and bit file. First, from primary and then to golden if the CRC check fails for the primary set.

The main firmware is stored in the external SPI flash. During booting, the boot loader copies the instruction code from the external flash to DDR4. Further, it sets up the ISR function pointer to this DDR4 memory address through the memory controller.

The MPMC IP works on top of LPDDR4 memory controller to write the instruction code to a specific DDR4 memory location.

SGDMA IP is used as data mover. It converts incoming AXI Stream from TSE MAC IP into AXI4 and sends to MPMC. Similarly, it converts the AXI4 interface coming from MPMC into AXI Stream and sends it to TSE MAC IP.

2.2.1.4. Memory Map

Table 2.1 shows the memory map of GSRD.

Table 2.1. GSRD Memory Map

Base Address	End Address	Range (Bytes)	Range (Bytes in hex)	Size (Kbytes)	Block
00300000	0037FFFF	512000	80000	512	SPI FLASH CONTROLLER
00000000	0003FFFF	256000	40000	256	CPU Data RAM
10000000	10000FFF	4096	1000	4	GPIO
10001000	10004FFF	16384	4000	16	TSE MAC
10090000	10090FFF	4096	1000	4	UART
10092000	10092FFF	4096	1000	4	LPDDR4 APB
10093000	10093FFF	4096	1000	4	SGDMA
1009B000	1009BFFF	4096	1000	4	FGPA CONFIG APB
10110000	5010FFFF	1073741824	40000000	1048576	LPDDR4 AXI
F2000000	F20FFFFF	1048576	100000	1024	CLINT (CPU)
FC000000	FC3FFFFF	4194304	400000	4096	PLIC (CPU)
F0000400	FFFFFFFF	262144000	FA00000	256000	RESERVED (CPU)

2.2.2. Lattice Main System 3.1 Architecture

The main system of Automate stack leverages GSRD as the basic building block and adds additional IP that are required to enable communication to nodes and host PC. For details about the GSRD, refer to the GHRD/GSRD Reference Design documents.

2.2.2.1. Lattice Main System 3.1 Architecture

The main system architecture is shown in Figure 2.4.The following are the components of the Main System:

- Processors
  - RISC-V CPU running FreeRTOS: Real time embedded operating system.
  - CNN Coprocessor Unit: AI/ML powered predictive maintenance system.
- Memory and Storage
  - SGDMA
  - FIFO DMA
  - RAM (ISR/Data): Shared memory
  - QSPI Memory Controller with pre-fetch buffer (SPI Flash Controller)
- Communication Interfaces
  - AXI Interconnect: Communicates between modules in the main system.
  - AXI2APB Bridge: Communicates to modules in the main system which are not AXI enabled.
  - MPMC: High-speed, DDR subsystem to handle multiple data streams simultaneously.
  - UDP IP: Handles OPC UA communications (in the form of UADP packets) to and from the host PC.
  - UART: Sends debug prints and information to a terminal.
  - EtherConnect: Handles connection to the node system Ethernet connection to the host PC.

The main system runs at a CPU frequency of 100 MHz. The ethernet MAC protocol runs at 125 MHz. The DDR Interface runs at 133 MHz.

For the best performance and nearly deterministic latency, the EtherConnect port supports one physical interface and a chain of up to eight nodes.

The Multiport Extension, GPIO, and EtherConnect modules can generate interrupts to the RISC-V CPU.

The Main System's modules are connected to each other with an AXI4 bus interface. The RISC-V CPU, CNN Coprocessor Unit, FIFO DMA, EtherConnect, SPI Flash Controller, Multiport Extension, and AXI2APB bridge all have an AXI4 interface and can use this bus directly. The UDP IP, SGMII TSE MAC Wrapper, Multiport Extension, and GPIO use the AXI2APB bridge and APB interconnect to access the AXI4 bus.

To avoid memory contention, Port S0 of the Data RAM is used exclusively for RISC-V CPU access. The CNN Coprocessor Unit and FIFO DMA access Port S1 of the Data RAM.

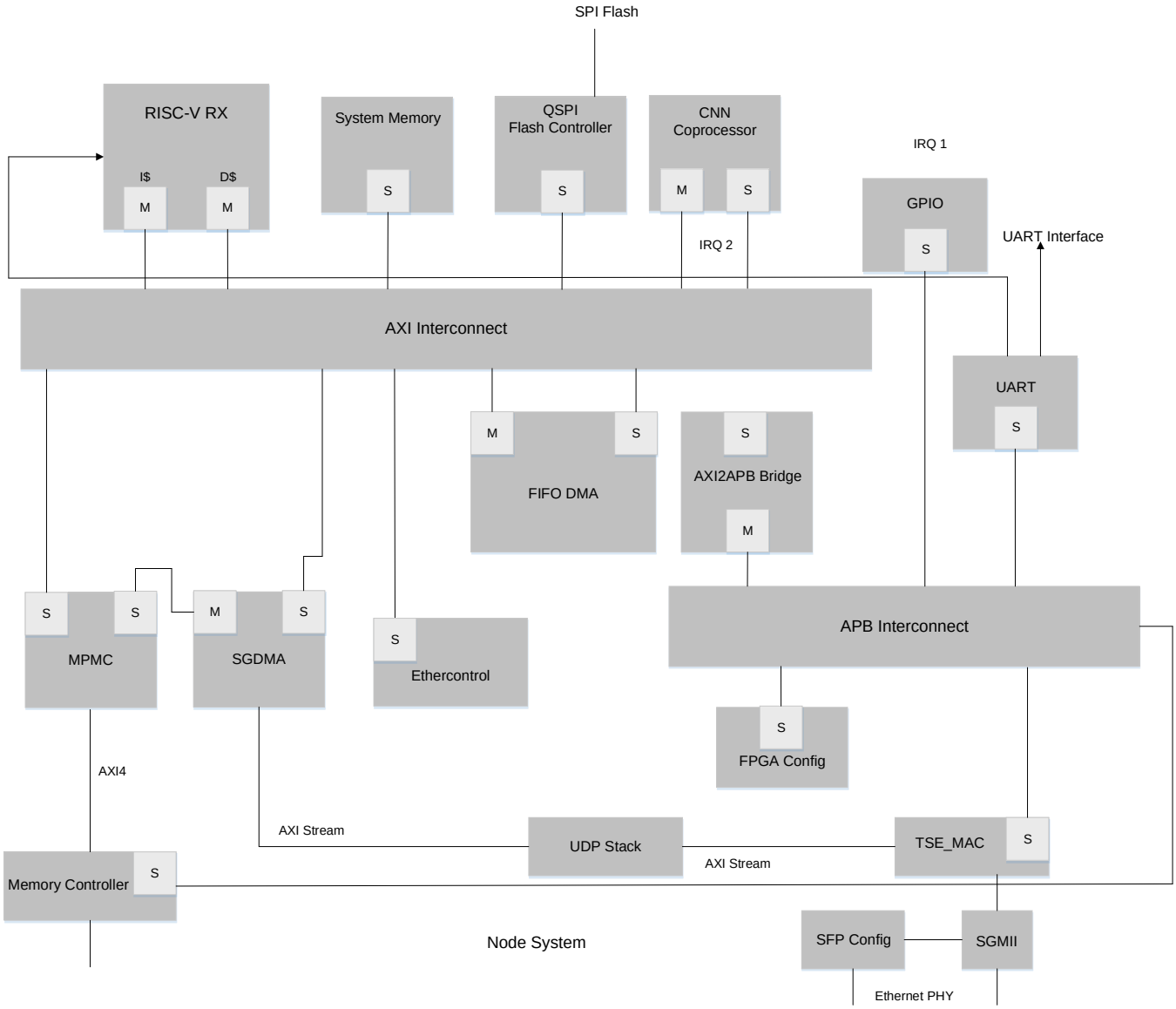

Figure 2.4. Main System Architecture

2.2.2.2. Main System Data Flow

The data flow from OPCUA Publisher (Host PC) to OPCUA Subscriber (Main board) and vice versa is shown in Figure 2.5. The host PC sends motor control commands using the Automate user interface to the main system over Ethernet cable. The host PC sends the UADP packet, and it is received by the main system using the SGMII, TSE MAC and UDP IP. The UDP IP parses this data and sends it to the SGDMA IP, which acts as a data mover. It converts incoming AXI Stream from UDP Stack IP into AXI4 data and sends to MPMC. Similarly, it converts the AXI4 data coming from MPMC into the AXI Stream and sends it to the UDP Stack IP. The MPMC IP sends this data to the LPDDR4 Memory controller which writes data into the LPDDR4 memory.

The Main System is an OPC UA publisher and an OPC UA subscriber. All OPC UA messages are sent as UADP packets. When the main system receives a UADP packet through Ethernet, the EtherConnect main module receives and sends its payload to the UDP IP. The UADP payload is written to the LPDDR4 and a CPU interrupt is triggered. The CPU then sends a read request to the Multiport Extension (which contains the LPDDR4). The packets are stored in a FIFO.

The CPU fetches data from the FIFO and passes it to the OPC UA software module. The OPC UA software module decodes the UADP packet and extracts an RFL command. RFL packets are created and sent to the node system over EtherConnect interface, which performs packetization and sends them downstream until they reach their destination node system. The Node System executes the command contained in the RFL packet.

The process for reading or writing to a peripheral connected to a node system through SPI, I^2C, or UART/Modbus is the same as reading from or writing to the motor control IP of the node system.

The Main System CPU gathers predictive maintenance data from downstream Node Systems through EtherConnect and sends to the host an OPC UA UDP packet through Ethernet. The Main System CPU also sends data to UART through an APB bus interface.

Alternatively, EtherConnect can send downstream data to FIFO DMA through its FIFO port and FIFO DMA can write the data-to-data RAM. At the end of every predictive maintenance cycle in SW running on main system, an update is sent to the host through Ethernet.

RISC-V RX can also communicate with various peripherals connected to nodes through SPI/I^2C/UART interfaces other than motor through host commands.

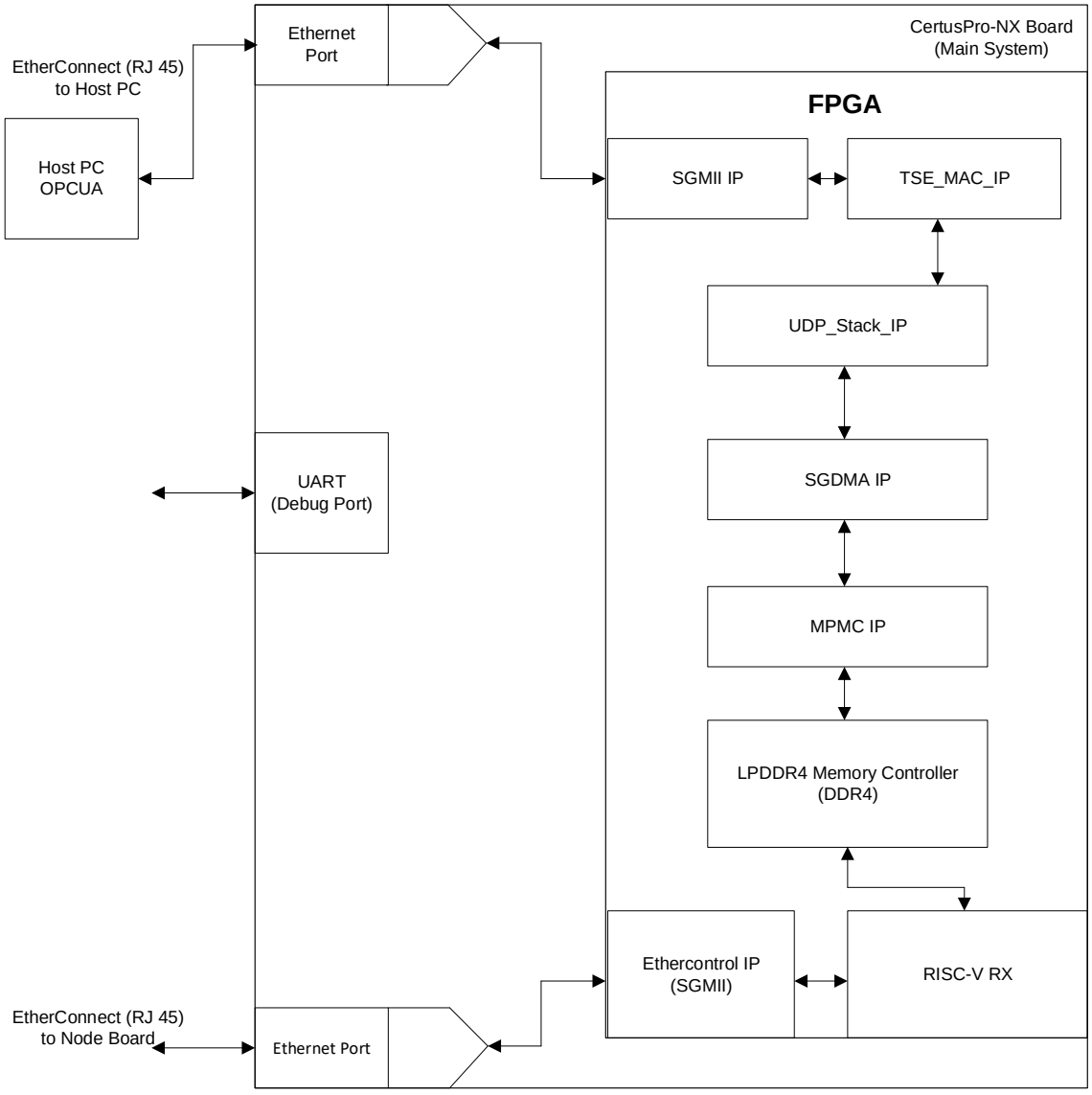

Figure 2.5. Client to Server Data Flow

2.2.2.3. Memory Map

Table 2.2 shows the memory map of the main system 3.1.

Table 2.2. Main System Memory Map

Base Address	End Address	Range (Bytes)	Range (Bytes in hex)	Size (Kbytes)	Block
00300000	0037FFFF	512000	80000	512	SPI FLASH CONTROLLER
00000000	0003FFFF	256000	40000	256	CPU Data RAM
10000000	10000FFF	4096	1000	4	GPIO
10001000	10004FFF	16384	4000	16	TSE MAC
10090000	10090FFF	4096	1000	4	UART
10092000	10092FFF	4096	1000	4	LPDDR4 Mem Controller APB
10093000	10093FFF	4096	1000	4	SGDMA
1009B000	1009BFFF	4096	1000	4	FPGA CONFIG APB
10100000	10107FFF	32768	8000	32	FIFO DMA
10108000	1010FFFF	32768	8000	32	EtherConnect
100A0000	100A0FFF	4096	1000	4	CNN Coprocessor
10110000	5010FFFF	1073741824	40000000	1048576	LPDDR4 AXI
F2000000	F20FFFFF	1048576	100000	1024	CLINT (CPU)
FC000000	FC3FFFFF	4194304	400000	4096	PLIC (CPU)
F0000400	FFFFFFFF	262144000	FA00000	256000	RESERVED (CPU)

2.2.3. Node System

The Node System architecture, shown in Figure 2.6. The following are the components of the Node System:

- Processors/Controllers
  - RISC-V CPU
  - Motor Control and PDM Data Collector
- Memory and Storage
  - FIFO DMA
  - RAM (ISR and Data): System Memory
  - QSPI Memory Controller with Prefetch buffer (SPI Flash Controller)
- Communication interfaces:
  - EtherConnect: Two-way communication with host system and with next node system in the chain.
  - AHBL2APB Bridge: Handles interface conversion between AHBL and APB.
  - SPI Manager: Communicates with peripherals through SPI interface.
  - I^2C Manager: Communicates with peripherals through I^2C interface.
  - UART: Communicates with peripherals through UART/Modbus interface.

The node system runs at a CPU frequency of 75 MHz, while the Ethernet protocol runs at 125 MHz.

To avoid memory contention, Port S0 of the Data RAM is used exclusively for RISC-V CPU access. The CNN Coprocessor Unit and FIFO DMA access port S1 of the Data RAM.

Note: Physically, there is only one piece of shared memory but with two independent ports. In the memory map, S0 is assigned with a lower base address and S1 is assigned with a higher base address. In real terms, these refer to the same physical address. The two different address spaces for S0 and S1 allow the AXI4 Interconnect to route the transaction to the right port.

The Motor Control and PDM Data Collector has two AHBL ports, S0 and S1. Port S0 is used to access the Motor Control and PDM registers, while port S1 is used to access the data collected by PDM Data Collector.

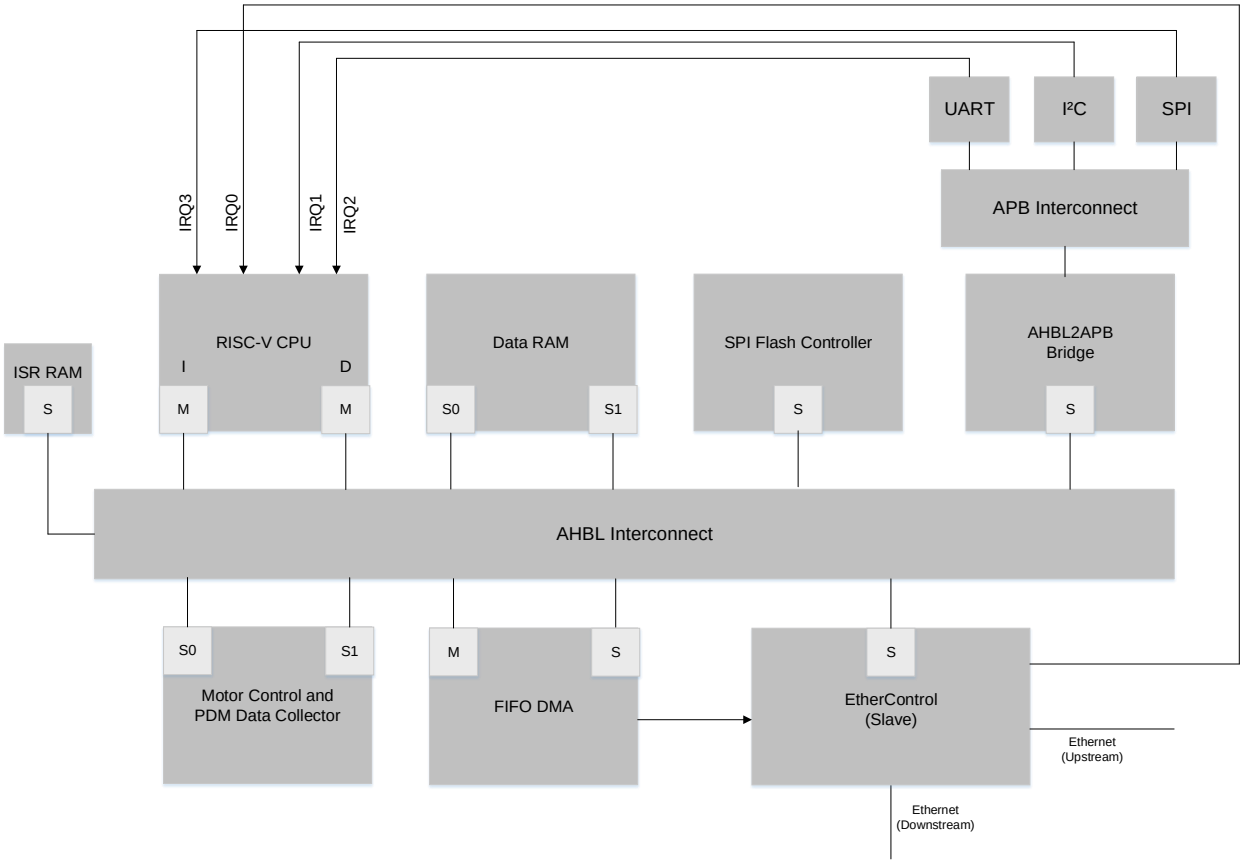

Figure 2.6. Node System Architecture

2.2.3.1. Data Flow

The main firmware is stored in the external SPI flash. The ISR RAM contains the initial boot code for RISC-V as well as interrupt service routines (ISRs) and other performance-critical functions.

The RISC-V CPU can stream its firmware from SPI Flash Controller through its AHBL port into the Instruction RAM The CPU can access data from the Data RAM, access the register file inside EtherConnect, and the control the registers of the FIFO DMA and QSPI Memory Controller. Both the RISC-V CPU and the FIFO DMA can move the data stored at the register file inside the EtherConnect to the Motor Control block. These can also move the data collected by the PDM Data Collector back to the EtherConnect to be sent back to the main system through the Ethernet upstream port.

The EtherConnect's protocol layer supports frame/packet type 10, which enables the system to enhance performance while fetching bulk data. See the EtherConnect user guide for more details.

2.2.3.2. Memory Map

The Node System memory map is listed in Table 2.3.

Table 2.3. Memory Map of Node System

Base Address	End Address	Range (Bytes)	Range (Bytes in hex)	Size (Kbytes)	Block
00190000	00197FFF	32768	8000	32	CPU instruction RAM
00080000	000807FF	2048	800	2	CPU PIC TIMER
00080800	000BFFFF	260096	3F800	254	RESERVED
000C0000	000FFFFF	262144	40000	256	CPU Data RAM Port S0 base address: 0x000C0000 Port S1 base address: 0x000E0000

Base Address	End Address	Range (Bytes)	Range (Bytes in hex)	Size (Kbytes)	Block
00100000	00107FFF	32768	8000	32	FIFO DMA
00108000	0010FFFF	32768	8000	32	EtherConnect
00110000	0017FFFF	458752	70000	448	RESERVED
00000000	0007FFFF	512000	7D000	512	SPI Flash Controller
001864000	001867FF	1024	400	1	UART
00184000	00185FFF	8192	2000	8	Motor Control and PDM Data Collector Port S0 base address: 0x00184000 Port S1 base address: 0x00185000
00186000	00FFFFFF	15179776	E7A000	14824	RESERVED
01400000	01FFFFFF	16777216	1000000	16384	External SPI flash
001868000	00186BFF	1024	400	1	SPI Master
00186000	001863FF	1024	400	1	I2C Master

2.3. EtherConnect IP Design Details

2.3.1. Overview

The EtherConnect block is used by both the Main System and the Node System. The Verilog parameter, *SYSTEM_TYPE*, sets this block as either Main System or Node System upon instantiation. The Main System has two Ethernet downstream ports and no upstream port. The Node System has one Ethernet upstream port and one Ethernet downstream port. The Ethernet downstream port can be disabled for the last Node System in the chain.

The SYSTEM_TYPE parameter also selects the Input or Output FIFO interface. In the main system, the EtherConnect IP has an output FIFO interface to send bulk data to the DMA FIFO block. In the node system, the EtherConnect block has an input FIFO interface to receive bulk data from the PDM Data Collector through the DMA FIFO module. The EtherConnect block uses an AXI4 interface along with a FIFO interface for bulk data.

The Sync Pulse generator block is available in the EtherConnect Main System only. It is used to generate pulse for the dynamic synchronization of nodes.

The EtherConnect IP block is designed for communication between two boards for information transfer and it is designed based on the Ether Connect protocol. The physical interface can support speed up to 1 Gbps (125 MHz clock). It supports both SGMII and RGMII interfaces in the physical layer as well as the SFP interface.

As a manager, it works in four layers:

- AHBL layer – used to have a connection with the RISC-V CPU and the register interface.
- Application layer – consists of data generation and sampling layers for the application.
- Protocol layer – used to transmit and receive Ether Connect packets.
- Physical layer - transfers data with protocol layer in GMII format and has RGMII and SGMII blocks to transmit or receive data over physical channels in RGMII or SGMII format.

The frame structure on the protocol level is shown in Figure 2.7.

Preamble	55_55_55	3octets
Sfd	d5	1octet
Sequence num	8'hxx	1octet
Pkt_type Slave number	2'hxx 6'hxx	1octet
Slave data len	8'hxx	1octet
Res Slave ID	2'hxx 6'hxx	1octet
Slave0 data	8'hxx ... ... 8'hxx	32octets
Slave1 data	8'hxx ... ... 8'hxx	32octets
FCS	8'hxx 8'hxx	2octets
Error indication	8'hxx	1octet

Figure 2.7. Packet Structure

2.3.1.1. Normal Packet

The changes are made for normal packet only. The request and response packet structure of old version is described below.

The normal frame type (00) has three types of packets:

- Packet type 01 – Configuration
- Packet type 02 – Status
- Packet type 03 – PDM

For configuration type packet, the data written in FIFO present in the application layer is as follows:

- The first 4 bytes indicate the packet type.
- The next 4 bytes indicate the node address.
- After that the data is sent in the next 4 bytes.
- The subsequent content of the packet is dummy data (00) for 52 bytes or in a generalized case: *(NODE_DATA_LENGTH - 12)*.

For the status type packet, the data written in FIFO present in application layer is as follows:

- The first 4 bytes indicate the packet type.
- The next 4 bytes indicate the node address.
- The subsequent content of the packet is dummy data (00) for 56 bytes or in a generalized case: *(NODE_DATA_LENGTH - 8)*.
- The response of the status packet is 32-bit status value, which is fetched from a register (*CH1_BASE_ADDR + 0x100*).

For the PDM type packet, the data written in FIFO present in application layer is as follows:

- The first 4 bytes indicate the packet type.
- The next 4 bytes indicate the node address.
- After that, the data is sent in the next 4 bytes.
- Next 4 bytes in the packet indicate the data length. The subsequent content of the packet is dummy data (00) for 48 bytes or in a generalized case: *(NODE_DATA_LENGTH - 16)*.
- The response of PDM packet is 4 kB PDM data, which can be stored in FIFO or can be send out through AXI Bus based on the value of CONTROL register.

2.3.2. Architecture

According to the new architecture, exchange of data between the main and node system changes. The request packet coming from the RISC-V CPU to the main system passes to the node system, as done earlier, but the response coming from the node system to the main system changes. Instead of reading the 32-bit status from the register, the complete response of status packet is written in the FIFO, which can be read by RISC-V using the register BASE_ADDR + 0x2C.

2.3.2.1. Main System

The protocol layer and physical layer remain as it is in the new version. The changes are done in *axi_subordinate_0_bus_control* for register addition and *ether_connect_manager_data_capture* module only for the response received from node. One FIFO is introduced to store the response of status packet. Depth of FIFO = max node data length × max number of nodes.

One local parameter, ETHER_EXTEN_EN, decides whether sampling of response in the application capture module is done using the old architecture or the new architecture.

2.3.2.2. Node System

In the node side, two new FIFOs are added for storing complete sampled data of the configuration packet and status packet. Each node samples its own data. For sampling configuration packet, interrupt is generated to indicate that the configuration is applied to the motor.

For status packet, the status of the node motor is stored in the FIFO2 and the signal is generated that the complete packet is received in the FIFO and is ready to send response.

2.3.3. Register Map

The register map of EtherConnect IP remains the same, except that one register is added to read the response of status the packet, which is highlighted in Table 2.4 and one register, the Node Motor Status Register, is removed .The data is read from the status FIFO when AXI read command is issued for address BASE + 0x2C.

Table 2.4. EtherConnect IP Global Registers

Ether Control Register Name	Register Function	Base Address (0x10108000)	Access
DMACTR_R	DMA FIFO Enable/AXI Disable Register	Base + 0x00	Read/Write
PHLNK_R	Phy Link Status Register	Base + 0x04	Read
NDACT_R	Active Nodes Register	Base + 0x08	Read
FSRPDM_R	FIFO Status Register for PDM Data CDC	Base + 0x0C	Read
ETHINTR_R	Interrupt Poll Register	Base + 0x10	Read
CLRCVD_R	Clear Interrupt Received Register	Base + 0x14	Read/Write
TX_ALL_STRT_R	Transaction start for all chains	Base + 0x18	Read/Write
DTOUT_R	Node Response PDM Data Register	Base + 0x1C	Read
IP_STATUS_R	IP Busy Status	Base + 0x20	Read/Write
AXI_TOUT_R	AXI Bus Timeout Count Register	Base + 0x28	Write
ND_STAT	Node Status Response	Base + 0x2C	Read

Table 2.5. EtherConnect IP Chain 1 Registers

Ether Control Register Name	Register Function	Base Address (0x10108100)	Access
TXSTR_R_1	Start Transaction Register	Base + 0x00	Read/Write
PKTHD_R_1	Packet Head Register	Base + 0x04	Read/Write
FRNUM_R_1	Frame Number Register	Base + 0x08	Read/Write
NDCNT_R_1	Number of Node Register	Base + 0x0C	Read/Write
NDLN_R_1	Node Data Length Register	Base + 0x10	Read/Write
MTDT_R_1	Node Request Data Burst Register	Base + 0x14	Read/Write

Ether Control Register Name	Register Function	Base Address (0x10108100)	Access
RQDT_R_1	Node Request Type Register	Base + 0x18	Read/Write
RQAD_R_1	Node Address Register	Base + 0x1C	Read/Write
CRCNT_R_1	CRC Count Register	Base + 0x20	Read
INTR_R_1	Interrupt Info Register	Base + 0x24	Read
FSRREQD_R_1	FIFO Status Register Request Data	Base + 0x28	Read
DLY_R_1	Node Delay Register	Base + 0x200 to 0x2FC	Read

2.4. FIFO DMA

This block has two FIFO interfaces, one is active when it is used in the main system to collect the PDM data received by the EtherConnnect manager Bus 0. The other interface is active for node and has the PDM data from the motor control data collector block.

This block also has an AXI4 subordinate and manager interface. The register space for this block is shown in Table 2.6.

The AXI4 Subordinate interface is used to control DMA operations by external manager (which is CPU) and AXI4 manager interface is used to perform for DMA operations. For more information on this IP, see the FIFO DMA user guide.

Table 2.6. FIFO DMA Register Map

Register Name	Register Function	Address	Access
CNTR	FIFO DMA Control Register	Base + 0x00	Read/Write
DEST_BASE_ADDR	Destination Base Address Register	Base + 0x04	Read/Write
DEST_END_ADDR	Destination End Address Register	Base + 0x08	Read/Write
STATUS	Write Status Register	Base + 0x0C	Read
STATUS_RD	Read Status Register	Base + 0x10	Read

Table 2.7. FIFO DMA Control Registers

CNTR				Base +0x00
Byte	3	2	1	0
Name	CNTR			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

CNTR[0]: Used to control read operation.
CNTR[1]: Used to reset the destination register to destination base address.
CNTR[2-7]: Reserved

Table 2.8. DEST_BASE_ADDR Register

DEST_BASE_ADDR				Base +0x04
Byte	3	2	1	0
Name	DEST_BASE_ADDR			
Default	0	0	0	0
Access	R/W			

DEST_BASE_ADDR[31:0]: Base Address Location

Table 2.9. DEST_END_ADDR Register

DEST_END_ADDR				Base +0x08
Byte	3	2	1	0
Name	DEST_END_ADDR			
Default	0	0	0	0
Access	R/W			

DEST_END_ADDR[31:0]: END Address Location

Table 2.10. Write Status Register

STATUS				Base +0x0C
Byte	3	2	1	0
Name	STATUS			
Default	Reserved	Reserved	Reserved	0
Access	R			

STATUS[2:0]: Write Status
STATUS[3:31]: Reserved

Table 2.11. Read Status Register

STATUS_RD				Base +0x1C
Byte	3	2	1	0
Name	STATUS_RD			
Default	Reserved	Reserved	Reserved	0
Access	R			

STATUS_RD[2:0]: Read Status
STATUS_RD[3:31]: Reserved

2.5. UDP Stack

The UDP Stack IP is specialized toward data transmission and reception over the internet. The UDP Protocol helps to establish a low-latency and loss-tolerating connections established over the network. Flexibility is ensured through run-time programmability of all the required network parameters (local, destination and gateway IP addresses, UDP ports, and MAC address).

The main block for the UDP protocol implementation is UDP Rx and UDP Tx. Also, the IP core supports the essential Address Resolution Protocol (ARP) and NDP (Neighbor Discovery Protocol) for multiple access networks and ICMP (Internet Control Message Protocol) and ICMP6 (for IPv6) Echo Request and Response messages ("ping") to test network connectivity. This IP supports commonly used standard interfaces for configuration and data such as APB and AXI4 stream respectively.

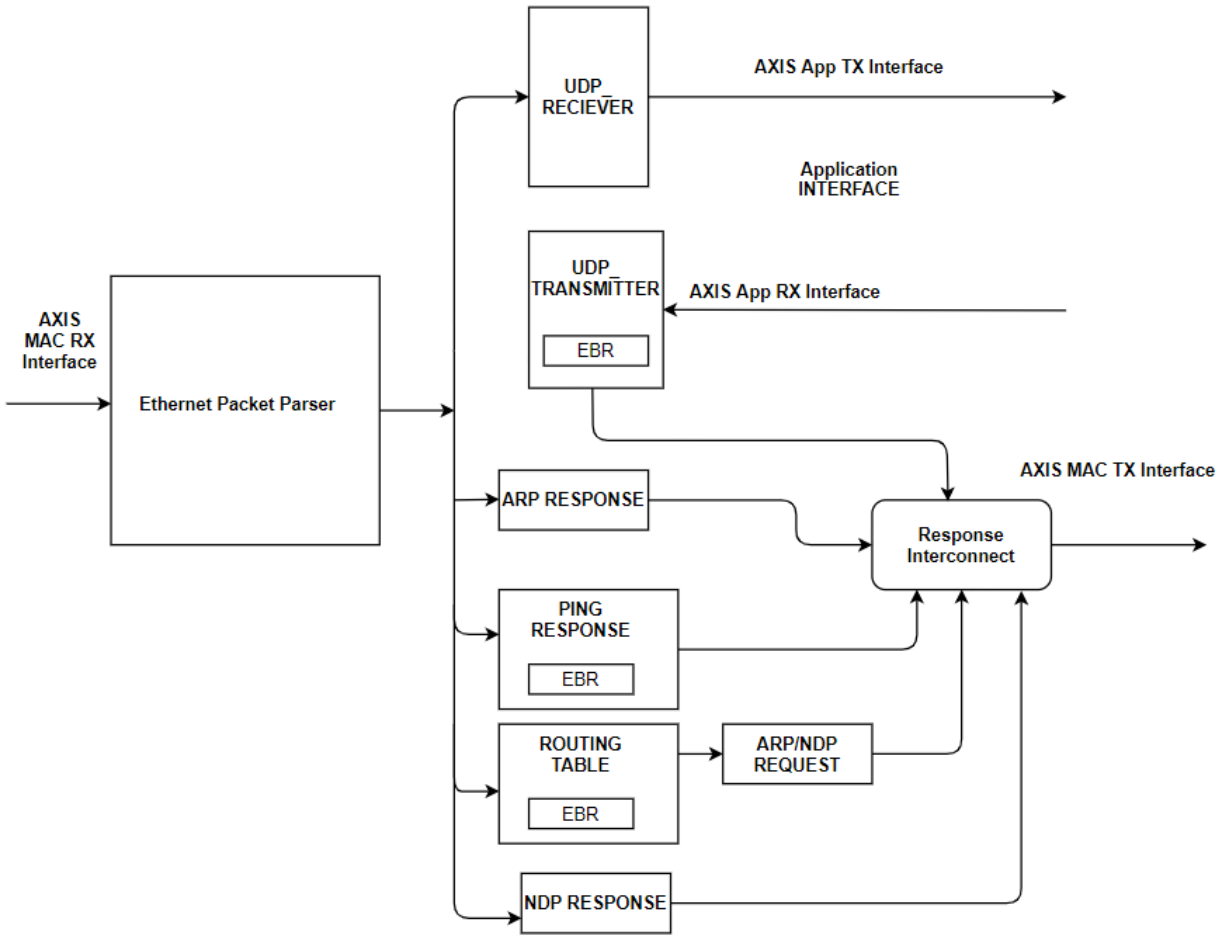

Figure 2.8. UDP Stack Top Level Architecture

To change the IPV4 address:

1. Open project on the Lattice Propel™ builder.
2. Double-click on the *udp_stack0_inst*.
3. To change the UDP destination IP address, update the CONFIG_TX_DST_IP_ADD macro. Here is an example IP address and the valid range: *c0a80102 to c0a801xx (xx could be 01 to FF values are in hex)*.

   To change the main system IP address, update the CONFIG_IPV4_ADD macro. Here is an example IP address and the valid range: *c0a80104 to c0a801xx (xx could be 01 to FF values are in hex)*.
4. Click **Generate**.

   Note: Make sure that the IPV4 and Destination IP address should be nearer/aligned to default gateway. These settings must be changed in the laptop/desktop IP address configuration settings mentioned in Appendix D.2 of the Automate Stack 3.1 Demo User Guide (FPGA-UG-02164). For more details, refer to C.4. Generating the Bit File to generate bitstream.

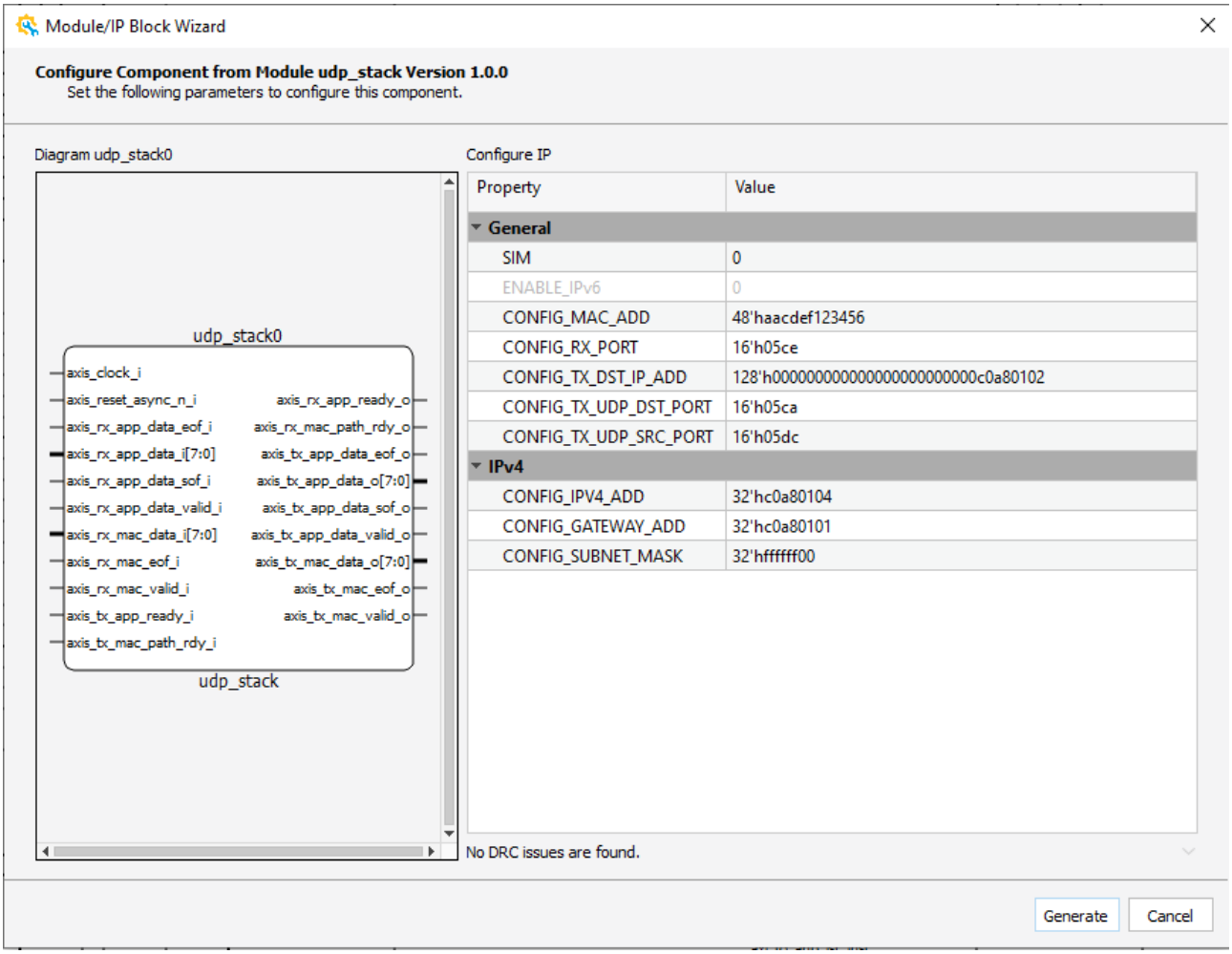

Figure 2.9. UDP Stack IPV4 address configuration

2.6. LPDDR4 Controller

The Lattice Semiconductor LPDDR4 Memory Controller for Nexus Devices provides a turnkey solution consisting of a controller, DDR PHY, and associated clocking and training logic to interface with LPDDR4 SDRAM. The IP Core is implemented in System Verilog HDL using the Lattice Radiant™ software integrated with the Lattice Synthesis Engine (LSE) and Synplify Pro® synthesis tools. The LPDDR4 Memory Controller simplifies the interfacing of CertusPro-NX and MachXO5T™-NX devices with external LPDDR4 memory for user applications.

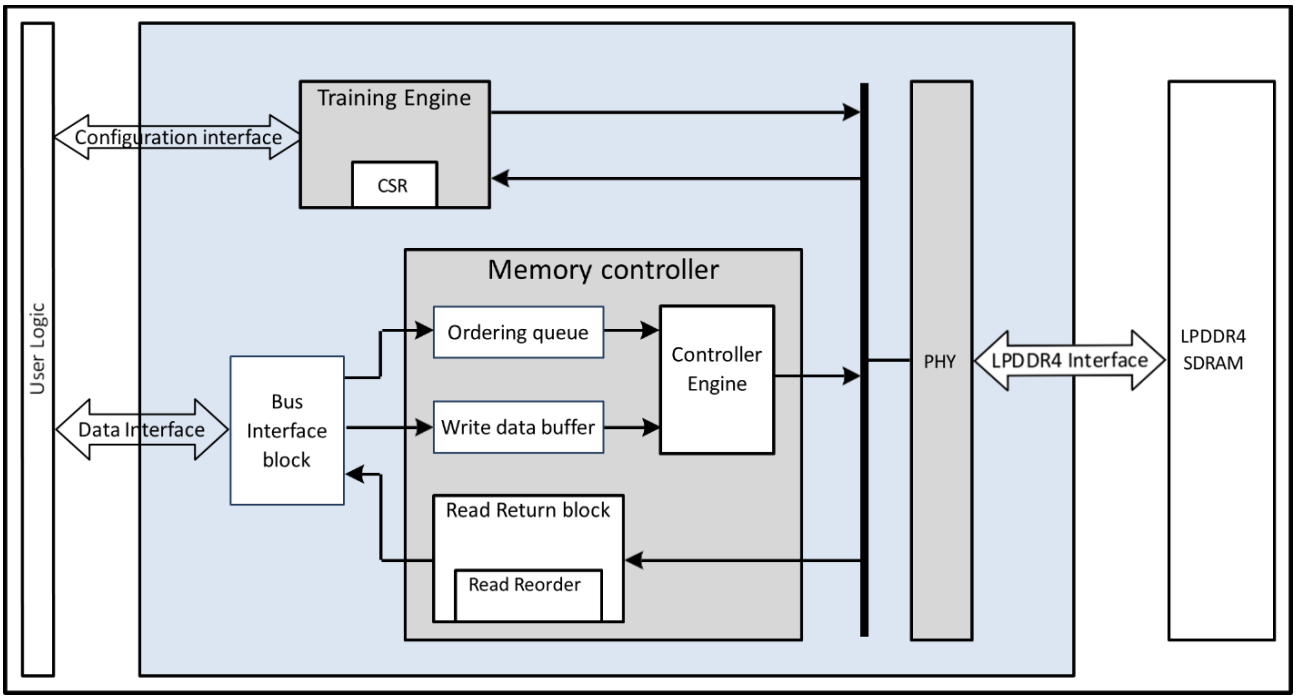

Figure 2.10. Memory Controller IP Core Functional Diagram

The data interface allows you to initiate LPDDR4 command/address/control and read/write operations to the external LPDDR4 SDRAM. The configuration interface provides access to the Training Engine and the Configuration Set Registers (CSRs), which configure the Memory Controller and perform the LPDDR4 training sequences. The LPDDR4 interface allows the selected Lattice FPGA to communicate with the external LPDDR4 memory.

The register map is shown in section 5 of the LPDDR4 IP User Guide. For more details, refer to LPDDR4 Memory Controller IP Core for Nexus Devices User Guide (FPGA-IPUG-02127).

2.7. QSPI Flash controller

A Quad Serial Peripheral Interface (QSPI) is a four-tri-state data line serial interface that is commonly used to program, erase, and read SPI Flash memories. QSPI enhances the throughput of a standard SPI by four times since four bits are transferred every clock cycle,

A Dual Serial Peripheral Interface (DSPI) uses two tri-state data lines and used to program, erase and read SPI Flash memories. DSPI performance is a comprise between QSPI and SPI since two bits are transferred every clock cycle.

The Lattice QSPI Flash Controller IP core supports SPI, DSPI and QSPI protocol. The design is implemented in Verilog HDL. It can be configured and generated using Lattice Propel™ Builder. It can be targeted to Nexus™ and Lattice Avant™ FPGA devices.

The QSPI Flash Controller IP Core allows the host inside the FPGA to communicate with multiple external SPI flash devices using either standard, extended dual/quad, dual or quad SPI protocols.

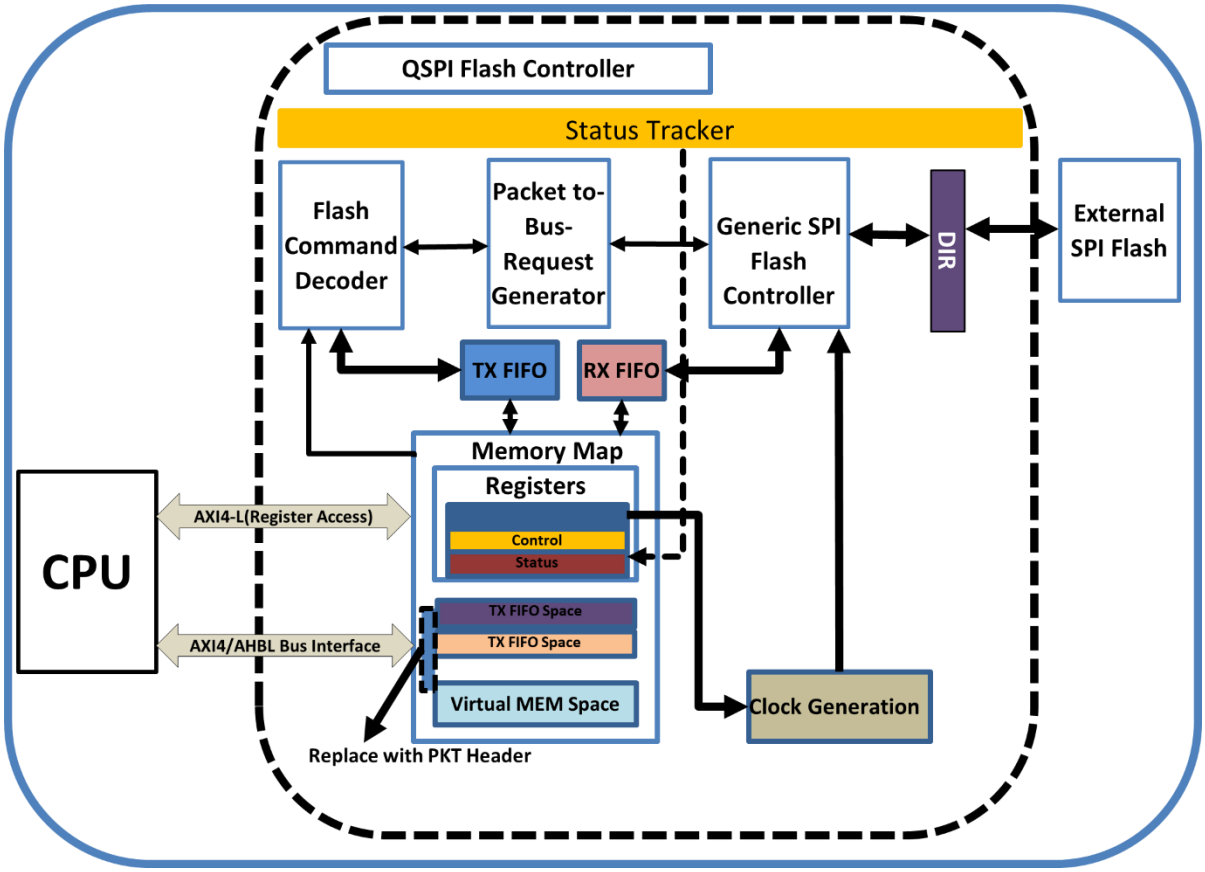

Figure 2.11. QSPI Flash Controller Block Diagram

The register map is shown in section 2.2 of the QSPI IP User Guide. For more details, refer to QSPI Flash Controller IP Core User Guide (FPGA-IPUG-02248).

2.8. Multi-Port Memory Controller IP Design Details

The MPMC acts an additional gateway to a separated memory controller IP (LPDDR4 memory controller in Automate 3.1).

The DRAM has a bus width of 32-bit while the AXI bus width of MPMC is programmable from 8, 16, 32, 64, 128, 256, and 512.

The input ports to the MPMC can be variable bus width and clock rates. When each port requests access to the DRAM, the MPMC must ensure the AXI interfaces on both side match throughput and no data drop. The MPMC must arbitrate which input port can access the MC/DRAM per configurable arbitration scheme, either round robin, fixed priority, or back pressure.

Each port has its own address range, per AXI4 memory mapped protocol. The address range is non-overlapping. The ports also have independent AXI bus width and operate at different clocks.

The functional block diagram is shown below:

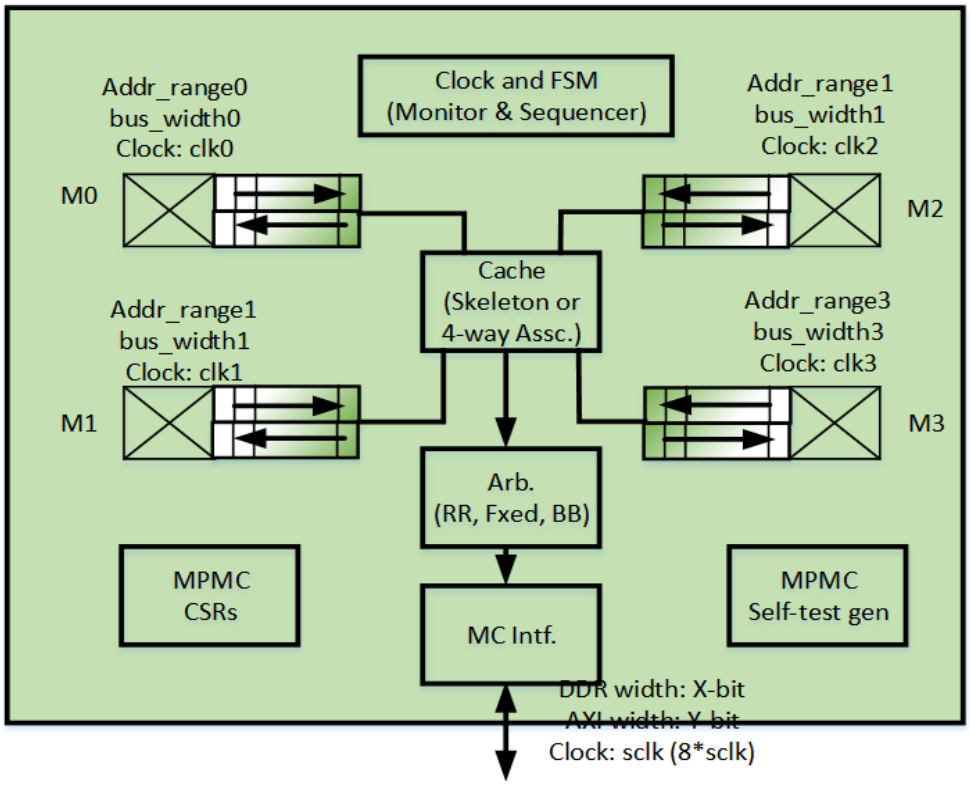

Figure 2.12. MPMC Top-level Block Diagram

The register map is shown in Table 2.12. The MPMC registers are configured one time through IP configuration user interface.

Table 2.12. MPMC Register Map

Offset	Register Name	Access Type	Description
0x00	DEVICE_FAMILY	RW	—
0x01	NUMBER_OF_PORTS	RW	—
0x02	PORT0_ADDRESS_START	RW	—
0x03	PORT0_ADDRESS_END	RW	—
0x04	PORT1_ADDRESS_START	RW	—
0x05	PORT1_ADDRESS_END	RW	—
0x06	PORT2_ADDRESS_START	RW	—
0x07	PORT2_ADDRESS_END	RW	—
0x08	PORT3_ADDRESS_START	RW	—
0x09	PORT3_ADDRESS_END	RW	—
0x0A	PORT0_BUF_SIZE	RW	—
0x0B	PORT0_BUF_ENABLE	RW	—
0x0C	PORT1_BUF_SIZE	RW	—
0x0D	PORT1_BUF_ENABLE	RW	—
0x0E	PORT2_BUF_SIZE	RW	—
0x0F	PORT2_BUF_ENABLE	RW	—

Offset	Register Name	Access Type	Description
0x10	PORT3_BUF_SIZE	RW	—
0x11	PORT3_BUF_ENABLE	RW	—
0x12	CACHE_WAY_SELECTION	RW	—
0x13	CACHE_LINE_BYTE	RW	—
0x14	CACHE_LINE_OFFSET	RW	—
0x15	ARB_SCHEME	RW	—
0x16	DRAM_FULL_CLOCK	RW	—
0x17	DRAM_BUS_WIDTH	RW	—
0x18	MC_AXI_BUS_WIDTH	RW	—
0x19	MC_AXI_CLOCK	RW	—
0x1A	TEST_PATTERN	RW	—
0x1B	TEST_START	RW	—
0x1C	TEST_STATUS	RW	—
0x1D	TEST_ERRORS	RW	—
0x1E	TEST_DONE	RW	—
0x1F	TEST_CONFIGURATION	RW	—
0x20	IRQ_STATUS	RW	—
0x21	MPMC_INIT_START	RW	—
0x22	MPMC_INIT_DONE	RW	—
0x23	MPMC_RESET	RW	—
0x24	MPMC_MC_RESET	RO	—
0x25	PORT0_TIMEOUT	RW	—
0x26	PORT1_TIMEOUT	RW	—
0x27	PORT2_TIMEOUT	RW	—
0x28	PORT3_TIMEOUT	RW	—
0x29	RESERVED	RW	—

2.9. Scatter Gather DMA IP Design Details

The Scatter Gather Direct Memory Access Controller (SGDMA) IP core provides access to the main memory independent of the processor, which offloads the processor intervention. The processor initiates transfer to SGDMAC and receives interrupt on completion of the transfer by DMA Engine.

The Lattice SGDMAC IP core implements a configurable, multi-channel, AHB Lite -compliant DMA controller with scatter-gather capability.

Direct Memory Access (DMA) is a technique for transferring blocks of data between system memory and peripherals without a processor (for example, system CPU) having to be involved in each transfer. DMA not only offloads a system processing elements but can transfer data at much higher rates than processor reads and writes.

Scatter-Gather DMA provides data transfers from one non-contiguous block of memory to another by means of a series of smaller contiguous-block transfers.

The Buffer Descriptors hold the necessary control information for data transfers:

- Source and destination buses and addresses.
- Amount of data to be transferred and maximum burst size.
- Addressing modes, bus sizes, transaction types, retry options, and others.

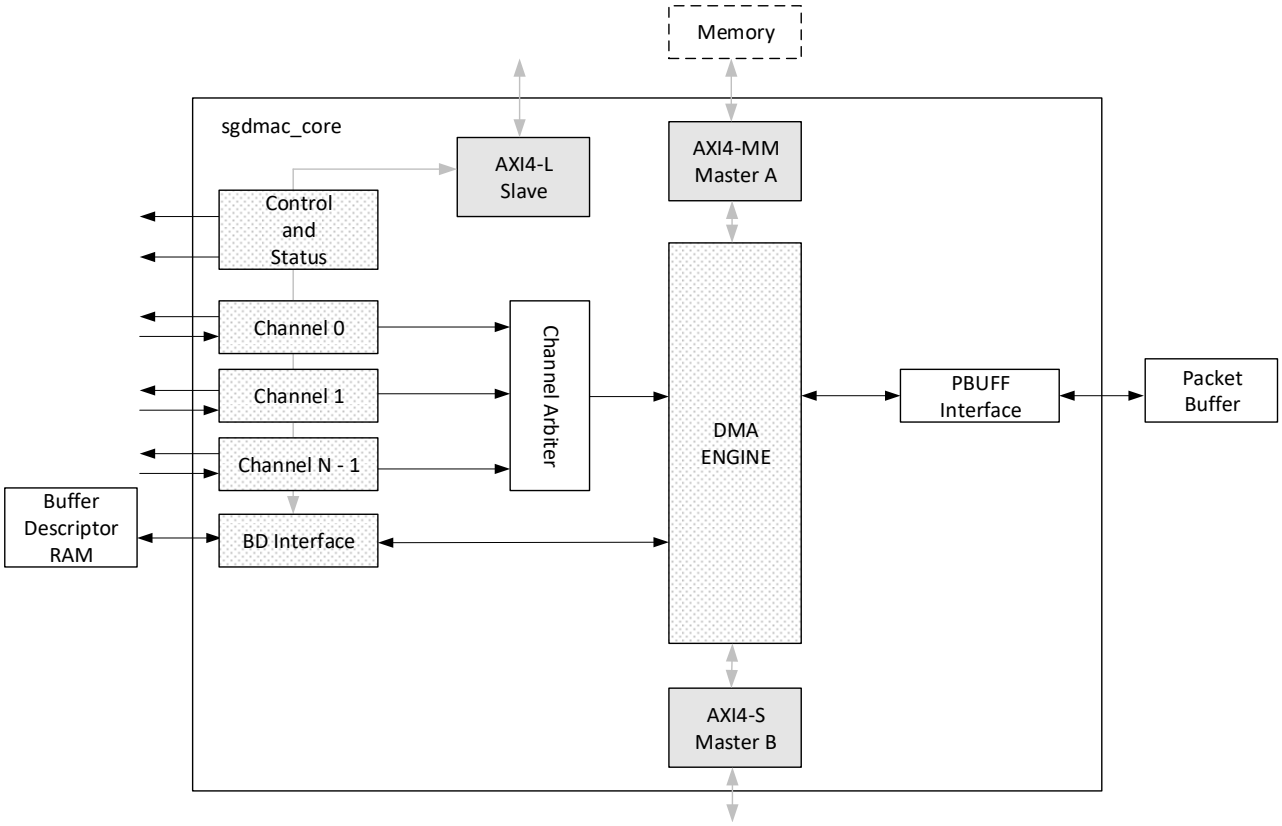

Figure 2.13. SGDMA IP Functional Diagram

The host can control the SGDMAC IP Core by writing to and reading from the configuration registers using the APB Interface.

The register map is listed in Table 2.13.

Table 2.13. Register Map of SGDMA IP

Offset	Register Name	Access Type	Description
0	IPID	R	IP identification register
4	IPVER	R	IP version register
8	GCONTROL	RW	Global control register
C	GSTATUS	RW	Global status register
10	GEVENT	RW	Global channel event register and mask
14	GERROR	RW	Global channel error register and mask
18	GARBITER	RW	Global arbiter control register
1c	GAUX	RW	Auxiliary inputs and outputs
N<<5 + 200	CONTROLN	RW	Control register
N<<5 + 204	STATUSN	RW	Status register
N<<5 + 208	CURSRCN	R	Current source register
N<<5 + 20c	CURDSTN	R	Current destination register
N<<5 + 210	CURXFERCNT	R	Current transfer count
N<<5 + 214	PBOFFSETN	RW	Packet buffer start address
X<<4 + 400	CONFIG0X	RW	Control register
X<<4 + 404	CONFIG1X	RW	Status register
X<<4 + 408	SRC_ADDRX	RW	Source address
X<<4 + 40c	DST_ADDRX	RW	Destination address

For more details, refer to SGDMA Controller IP Core User Guide (FPGA-IPUG-02131).

2.10. CNN Coprocessor Unit (CCU)

This block has an AXI4 Manager interface so that it can retrieve data directly from Data RAM or EtherConnect block. This block can also fetch data from UART. For example, after the host PC has processed the training data and come up with a new set of weights, the CCU can get the new weights through UART.

This block also has an AXI4 subordinate interface so that RISC-V CPU can control CNN Coprocessor Unit (CCU) through its registers.

Table 2.14. CNN Coprocessor Unit Registers

CCU Register Name	Register Function	Address	Access
PDMACR	CCU Control Register	Base + 0x00	Read/Write
PDMASR	CCU Status Register	Base + 0x04	Read
SIGSELR	Sign Select Configuration Register	Base + 0x08	Read/Write
INOFFSETCR	Input Offset Configuration Register	Base + 0x0C	Read/Write
FILOFFSETCR	Filter Offset Configuration Register	Base + 0x10	Read/Write
INDEPTHCR	Input Depth Configuration Register	Base + 0x14	Read/Write
INADDRCR	Input Data Address Configuration Register	Base + 0x18	Read/Write
FILADDRCR	Filter Data Address Configuration Register	Base + 0x1C	Read/Write
ACCOUTR	CCU Output Register	Base + 0x20	Read

Table 2.15. CNN Coprocessor unit control register

PDMACR		Base + 0x00
Bits	Others	0
Name	Unused	START
Default	Unused	0
Access	Unused	R/W

START: Setting 1'b1 to this register triggers the start of CCU process

Table 2.16. CNN Coprocessor Unit Register

PDMASR		Base + 0x04
Bits	Others	0
Name	Unused	DONE
Default	Unused	0
Access	Unused	R

DONE :
1'b0: CCU process is NOT completed
1'b1: CCU process is completed

Table 2.17. Sign Select Configuration Register

SIGSELR		Base + 0x08
Bits	Others	0
Name	Unused	SIGN_SEL
Default	Unused	0
Access	Unused	R/W

SIGN_SEL: Sign selector of input and filter values
1'b0: Unsigned (TinyML HPD)
1'b1: Signed (ours)

Table 2.18. Input Offset Configuration Register

INOFFSETCR		Base + 0x0C
Bits	Others	8 : 0
Name	Unused	INPUT_OFFSET
Default	Unused	0
Access	Unused	R/W

INPUT_OFFSET: Input offset (2s complement - signed number [–256 ~ 255])

Table 2.19. Filter Offset Configuration Register

FILOFFSETCR		Base + 0x10
Bits	Others	8 : 0
Name	Unused	FILTER_OFFSET
Default	Unused	0
Access	Unused	R/W

FILTER_OFFSET: Filter offset (2s complement - signed number [–256 ~ 255])

Table 2.20. Filter Offset Configuration Register

FILOFFSETCR		Base + 0x10
Bits	Others	8: 0
Name	Unused	FILTER_OFFSET
Default	Unused	0
Access	Unused	R/W

Table 2.21. Input Depth Configuration Register

INDEPTHCR		Base + 0x14
Bits	Others	9: 0
Name	Unused	INPUT_DEPTH_BY_2_M1
Default	Unused	0
Access	Unused	R/W

INPUT_DEPTH_BY_2_M1: Input depth × 2 – 1 (0 ~ 1023); cover 512 depth.

Table 2.22. Input Data Address Configuration Register

INADDRCR		Base + 0x18
Bits	Others	16: 0
Name	Unused	INPUT_DATA_ADDR
Default	Unused	0
Access	Unused	R/W

INPUT_DATA_ADDR: Address to INPUT_DATA – start point of blob.

Table 2.23. Filter Data Address Configuration Register

FILADDRCR		Base + 0x1C
Bits	Others	16: 0
Name	Unused	FILTER_DATA_ADDR
Default	Unused	0
Access	Unused	R/W

FILTER_DATA_ADDR: Address to FILTER_DATA – start point of filter.

Table 2.24. CNN Coprocessor Unit Output Register

ACCOUTR		Base + 0x20
Bits	Others	31 : 0
Name	Unused	ACC_OUT
Default	Unused	0
Access	Unused	R

ACC_OUT: Accelerator output data

2.11. Motor Control and PDM Data Collector

This block has two AHBL subordinate interfaces that reside in the Node System. It provides direct control to motors through its logic and interface to power electronics. It also collects predictive maintenance data from the motors.

This block is used only in the Node Systems. The top level of the Node System has an AHBL wrapper which has two AHBL subordinate ports. Mainly it consists of Motor Control and Predictive Maintenance (MC/PDM) Registers, Motor Control logic, and PDM Data Collector as shown in Figure 2.14.

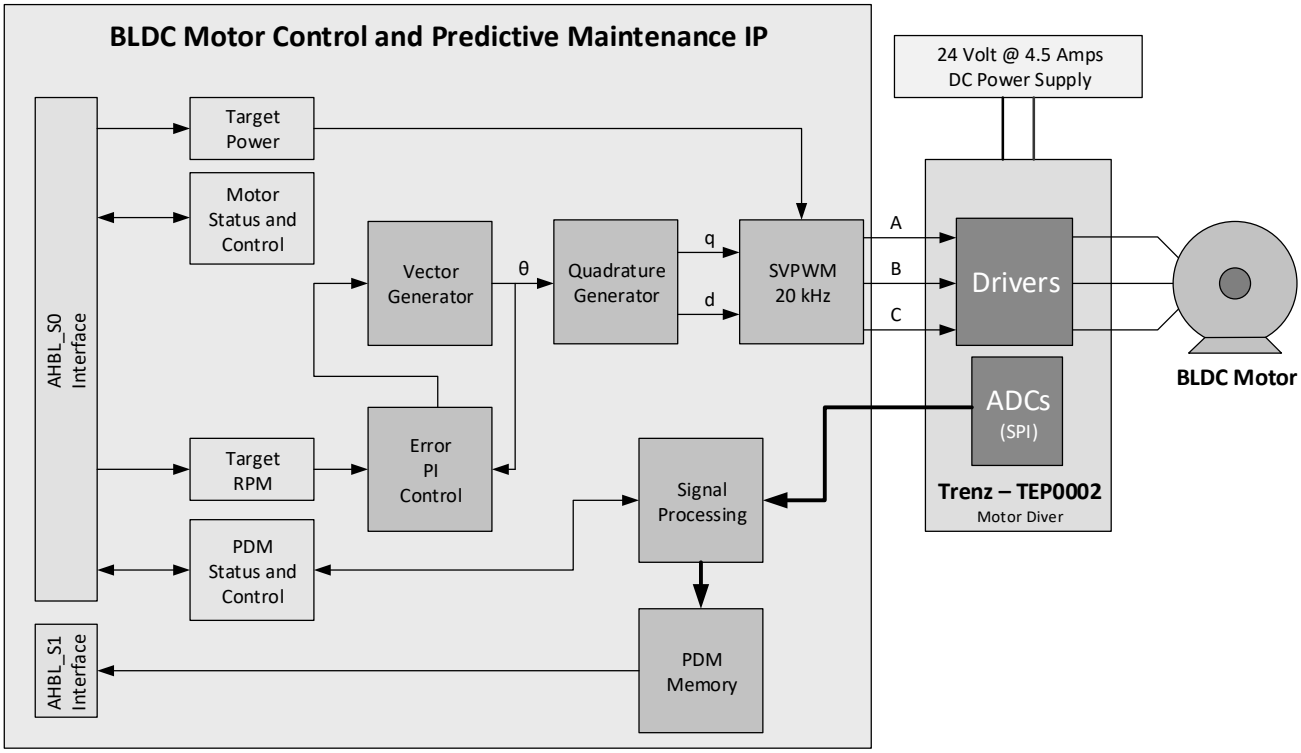

Figure 2.14. Motor Controller Interface with Motor

The Motor Control and PDM Registers interface with the AHB-L bus to configure, control, and monitor the Motor Control IP.

Table 2.25. Predictive Maintenance and Motor Control Registers

PDM/Motor Register Name	Register Function	Address	Access
MTRCR0	Motor Control Register 0 – Min RPM	Base + 0x00	Read/Write
MTRCR1	Motor Control Register 1 – Max RPM	Base + 0x04	Read/Write
MTRCR2	Motor Control Register 2 – RPM PI kI	Base + 0x08	Read/Write
MTRCR3	Motor Control Register 3 – RPM PI kP	Base + 0x0C	Read/Write
MTRCR4	Motor Control Register 4 – Torque PI kI	Base + 0x10	Read/Write
MTRCR5	Motor Control Register 5 – Torque PI kP	Base + 0x14	Read/Write
MTRCR6	Motor Control Register 6 – Sync Delay & Control	Base + 0x18	Read/Write
MTRCR7	Motor Control Register 7 – Target RPM	Base + 0x1C	Read/Write
MTRCR8	Motor Control Register 8 – Target Location	Base + 0x20	Read/Write
MTRCR9	Motor Control Register 9 – Location	Base + 0x24	Read/Write
MTRSR0	Motor Status Register 0 – RPM	Base + 0x28	Read
MTRSR1	Motor Status Register 1 – Limit SW & System Status	Base + 0x2C	Read
PDMCR0	Predictive Maintenance Control Register 0	Base + 0x30	Read/Write
PDMCR1	Predictive Maintenance Control Register 1	Base + 0x34	Read/Write
PDMSR	Predictive Maintenance Status Register	Base + 0x38	Read

PDM/Motor Register Name	Register Function	Address	Access
PDMDDR	Predictive Maintenance ADC Data Register	Base + 0x3C	Read
PDMQDR	Predictive Maintenance ADC Data Register	Base + 0x40	Read
BRDSW	DIP and Push Button Switches	Base + 0x50	Read
BRDLEDS	LEDs and 7-Segment	Base + 0x54	Read/Write

Table 2.26. Motor Control 0 – Minimum RPM

MTRCR0				Base + 0x00
Byte	3	2	1	0
Name	PI_DELAY	MTRPOLES	MINRPM	
Default	0	0	0	0
Access	R/W			

MTRCR0[15:0]: MINRPM – Minimum RPM is the initial open loop motor starting RPM. Valid values are 10 to (2^{16} -1).
MTRCR0[23:16]: MTRPOLES – Number of motor stator poles. Valid values are 1 to 255.
MTRCR0[31:24]: PI_DELAY – Is the RPM PI update rate. Valid values are 1 to 255.

Table 2.27. Motor Control 1 – Maximum RPM

MTRCR1				Base + 0x04
Byte	3	2	1	0
Name	tbd		MAXRPM	
Default	0	0	0	0
Access	R/W			

MTRCR1[15:0]: MAXRPM – Maximum RPM is the upper limit RPM. Valid values are MINRPM to (2^{16} -1).
MTRCR1[31:16]: TBD

Table 2.28. Motor Control 2 – RPM PI Control Loop Integrator Gain (kI)

MTRCR2				Base + 0x08
Byte	3	2	1	0
Name	RPMINT_MIN		RPMINTK	
Default	0	0	0	0
Access	R/W			

MTRCR2[15:0]: RPMINTK – Is the gain of the Integrator part of the RPM PI control loop. Valid values are 1 to (2^{16} -1).
MTRCR2[31:16]: RPMINT_MIN – Is the Integrator Anti-Windup Threshold. Valid values are 1 to (2^{16} -1).

Table 2.29. Motor Control 3 – RPM PI Control Loop Proportional Gain (kP)

MTRCR3				Base + 0x0C
Byte	3	2	1	0
Name	RPMINT_LIM		RPMPRPK	
Default	0	0	0	0
Access	R/W			

MTRCR3[15:0]: RPMPRPK – Is the gain of the Proportional part of the RPM PI control loop. Valid values are 1 to (2^{16} -1).
MTRCR3[31:16]: RPMINT_LIM – Is the Integrator Anti-Windup Clamp. Valid values are 1 to (2^{16} -1).

Table 2.30. Motor Control 4 – Torque PI Control Loop Integrator Gain (kI)

MTRCR4				Base + 0x10
Byte	3	2	1	0
Name	TRQINT_MIN		TRQINTK	
Default	0	0	0	0
Access	R/W			

MTRCR4[15:0]: TRQINTK – Is the gain of the Integrator part of the Torque PI control loop. Valid values are 1 to (2^{16} -1).
MTRCR4[31:16]: TRQINT_MIN – Is the Integrator Anti-Windup Threshold. Valid values are 1 to (2^{16} -1).

Table 2.31. Motor Control 5 – Torque PI Control Loop Proportional Gain (kP)

MTRCR5				Base + 0x14
Byte	3	2	1	0
Name	TRQINT_LIM		TRQPRPK	
Default	0	0	0	0
Access	R/W			

MTRCR5[15:0]: TRQPRPK – Motor Power or Torque PI Proportional Gain, depends on value of MTRCR6[2].
MTRCR6[2] = 0: Motor Power - valid values are 0 to 1023.
MTRCR6[2] = 1: Torque PI Proportional Gain - valid values are 1 to (216-1).[1]
MTRCR5[31:16]: TRQINT_LIM – Is the Integrator Anti-Windup Clamp. Valid values are 1 to (2^{16} -1).

Table 2.32. Motor Control 6 – Synchronization Delay and Control

MTRCR6				Base + 0x18
Byte	3	2	1	0
Name	MTRCTRL	SYNCDLY		
Default	0	0	0	0
Access	R/W			

MTRCR6[21:0]: SYNCDLY[1] – Is the Motor control delay to compensate for Ethernet daisy-chain and processing delay. Used to synchronize starting and stopping of multiple motors simultaneously. Valid values are 0 to (2^{22} -1).
MTRCR6[23:22]: MTRCTRL_SYNDLYSF[1] – Sync Delay Scale Factor
00 = Disable Sync Delay (single motor control or sync not used).
01 = Sync Delay Units is nano-seconds (10-9)
10 = Reserved
11 = Reserved
MTRCR6[24]: RESET_PI – Reset the RPM PI Control
0 = Normal Operation
1 = Force the output to match the input (zero input values force the output to default of 120 rpm)
MTRCR6[25]: STOP – Hold the Motor in Position
0 = Normal Operation
1 = Stop the motor rotation
MTRCR6[26]: TRQPI_MODE – Torque Control Mode controls how MTRCR5[15:0] : TRQPRPK is used:
0 = Open Loop Mode – TRQPRPK value specifies Motor Power.
1 = Closed Loop Mode – TRQPRPK value specifies the gain of the Proportional part of the Torque PI control loop.[1]
MTRCR6[27]: ESTOP – Emergency Stop
0 = Normal Operation.
1 = Engage E-Brakes without sync delay or MTR_ENGAGE.[1]

MTRCR6[28]: ENABLE – Enable Motor Drivers
0 = Disable Motor Drivers
1 = Enable Motor Drivers
MTRCR6[29]: MTR_MODE
0 = RPM Control – Slew to target RPM and continue to run until stop or change in RPM target
1 = Location Control – Rotate specified number of degrees or turns then stop. Ramp up from zero to Max RPM, run as needed, then ramp back down to zero.[1]
MTRCR6[30]: DIRECTION
0 = Clockwise Rotation
1 = Counter-Clockwise Rotation
MTRCR6[31]: ENGAGE – Sync Signal to latch all Control Registers from AHBL clock domain (50–100 MHz) to Motor clock domain (24–25 MHz). Write to all other control registers first (including this one with this bit off). Write to this register (read-modify-write) to set this bit. It can also be used to synchronize multiple nodes.
0 = No Updates to Motor or PDM Control registers.
1 = Transfer all control register from AHBL holding registers to Motor PDM active registers.

Table 2.33. Motor Control Register 7 – Target RPM

MTRCR7				Base + 0x1C
Byte	3	2	1	0
Name	TBD		TRGRPM	
Default	0	0	0	0
Access	R/W			

MTRCR7[15:0]: TRGRPM – Target RPM. Valid values are 0 to (2^{16} -1).
MTRCR7 [31:16]: tbd

Table 2.34. Motor Control Register 8 – Target Location

MTRCR8				Base + 0x20
Byte	3	2	1	0
Name	TRGLOC			
Default	0	0	0	0
Access	R/W			

MTRCR8[31:0]: TRGLOC – Target Location. Valid values are -2,147,483,648 (-2^{32}) to 2,147,483,647 (2^{32} -1).
Approximately 24.8 hours @ 4,000 RPM counting each degree.

Table 2.35. Motor Control Register 9 – Current Location

MTRCR9				Base + 0x24
Byte	3	2	1	0
Name	MTRLOC			
Default	0	0	0	0
Access	R			

MTRCR9[31:0]: MTRLOC – Motor Location. Valid values are -2,147,483,648 (-2^{32}) to 2,147,483,647 (2^{32} -1).

Table 2.36. Motor Status Register 0 – RPM

MTRSR0				Base + 0x28
Byte	3	2	1	0
Name	tbd		MTRSTRPM	
Default	0	0	0	0
Access	R			

MTRSR0[15:0]: MTRSTRPM – Current Motor RPM. Valid values are 0 to (2^{16} -1).
MTRSR0[31:16]: tbd.

Table 2.37. Motor Status Register 1

MTRSR1				Base + 0x2C
Byte	3	2	1	0
Name	MTRSR1			
Default	0	0	0	0
Access	R			

MTRSR1[0] : MTRSTR_MOV – Motor Moving
0 = Motor Stopped or coasting
1 = Motor Moving under control
MTRSR1[1]: ACCEL – Motor Accelerating
0 = Motor Not Accelerating
1 = Motor Accelerating
MTRSR1[2]: DECL - Motor Deaccelerating
0 = Motor Not Deaccelerating
1 = Motor Deaccelerating
MTRSR1[3]: RPM_LOCK - Motor At Target RPM
0 = Motor Not @ Target RPM
1 = Motor @ Target RPM
MTRSR1[4]: MTRSTR_STOP
0 = Motor not stopped
1 = Motor at zero RPM
MTRSR1[5]: MTRSTR_VLD_RPM
0 = RPM to Theta period calculation is still in process or invalid RPM request
1 = RPM to Theta period calculation is complete
MTRSR1[31:6]: tbd

Table 2.38. Predictive Maintenance Control Register 0

PDMCR0				Base + 0x30
Byte	3	2	1	0
Name	PDMCR0			
Default	0	0	0	0
Access	R/W			

PDMCR0[0]: START – Start PDM data collection.
0 = Collection not started
1 = Collection started
PDMCR0[1]: PKDTEN – PDM Normalization Peak Detect Enable
0 = PDM Peak Detect is Disabled
1 = PDM Peak Detect is Enabled
PDMCR0[2]: FOLDEN – Enable Single Folding of PDM data
0 = Single Fold disabled
1 = Single Fold enabled
PDMCR0[3]: 2FOLDEN – Enable Double Folding of PDM data. All PDM training data was captured using Double Folding.
0 = Double Folding disabled
1 = Double Folding enabled
PDMCR0[4]: CONTINUOUS – Collect data as long as START = 1.
0 = Fixed – Collect PDM data for set number of rotations
1 = Continuous – Collect PDM data continuously (counting rotations in status reg)

PDMCR0[5]: TBD
PDMCR0[6]: CALIB – ADC offset calibration
0 = Normal operation
1 = Calibrate ADC offsets (motor not running)
PDMCR0[7]: ADCH – ADC Channel Select for PDMDDR and PDMQDR registers
0 = ADC Channel = Amps
1 = ADC Channel = Volts
PDMCR0[15:8]: PREREVS – Pre Data Collection Revolutions
Number of Theta (Field Vector) revolutions to ignore before Data Collection. All PDM training data was captured using a value of 15.
PDMCR0[31:16]: DCREVS – Data Collection Revolutions
Theta (Field Vector) revolutions to capture PDM data (armature revs scale based on number of motor stator poles.
The motor used for training has 4-poles – 16 Theta rotations equate to four motor shaft rotations). Valid values 1 to 65,536. All PDM training data was captured using 200 rotations.

Table 2.39. Predictive Maintenance Control Register 1

PDMCR1				Base + 0x34
Byte	3	2	1	0
Name	PDMCR1			
Default	0	0	0	0
Access	R/W			

PDMCR1: TBD

Table 2.40. Predictive Maintenance Status Register

PDMSR				Base + 0x38
Byte	3	2	1	0
Name	PDMSR			
Default	0	0	0	0
Access	R			

PDMSR[0]: DONE – PDM activity status
0 = PDM is not done with collecting data
1 = PDM is done with collecting data
PDMSR[1]: BUSY – PDM activity status
0 = PDM is not active
1 = PDM is busy collecting data
PDMSR[2]: CAL_DONE – ADC Offset Calibration status
0 = Offset calibration is not done
1 = Offset calibration is done
PDMSR[3]: READY – PDM Data Collector status
0 = Not ready to collect data
1 = Ready to collect data
PDMSR[15:4]: TBD
PDMSR[31:16]: PDMSR_ROT – Current count of Theta rotations PDM data has been collected for.

Table 2.41. Predictive Maintenance Current/Voltage Data Register

PDMDDR				Base + 0x3C
Byte	3	2	1	0
Name	ADC1		ADC0	
Default	0	0	0	0
Access	R			

PDMDDR[15:0]: ADC0 Voltage or Current reading Phase A
PDMDDR[31:16]: ADC1 Voltage or Current reading Phase B

Table 2.42. Predictive Maintenance Current/Voltage Data Register

PDMQDR				Base + 0x40
Byte	3	2	1	0
Name	ADC3		ADC2	
Default	0	0	0	0
Access	R			

PDMQDR[15:0]: ADC2 Voltage or Current reading Phase C
PDMQDR[31:16]: ADC3 Voltage or Current reading of DC supply

Table 2.43. Versa Board Switch Status Register

BRDSW				Base + 0x50
Byte	3	2	1	0
Name	TBD	PMOD2	DIPSW	PBSW
Default	0	0	0	0
Access	R			

PBSW[0]: SW5 – Pushbutton 2
0 = Switch active (pressed)
1 = Switch inactive
PBSW[1]: SW3 – Pushbutton 1
0 = Switch active (pressed)
1 = Switch inactive
PBSW[2]: SW2 – Pushbutton 3
0 = Switch active (pressed)
1 = Switch inactive
PBSW[7:3]: n/c - undefined
DIPSW[3:0]: SW10 – DIP Switch
0 = Switch closed
1 = Switch open
DIPSW[7:4]: n/c – undefined
PMOD2[0]: J8 Pin 1 I/O
PMOD2[1]: J8 Pin 2 I/O
PMOD2[2]: J8 Pin 3 I/O
PMOD2[3]: J8 Pin 4 I/O
PMOD2[4]: J8 Pin 7 I/O
PMOD2[5]: J8 Pin 8 I/O
PMOD2[6]: J8 Pin 9 I/O
PMOD2[7]: J8 Pin 10 I/O

Table 2.44. Versa Board LED and PMOD Control Register

BRDLEDS				Base + 0x54
Byte	3	2	1	0
Name	PMOD2DIR	PMOD2	7SEG	LED
Default	0xF	0xF	0xF	0xF
Access	R/W			

LED[0]: LED D18 – 0 = On, 1 = Off
LED[1]: LED D19 – 0 = On, 1 = Off
LED[2]: LED D20 – 0 = On, 1 = Off
LED[3]: LED D21 – 0 = On, 1 = Off
LED[4]: LED D22 – 0 = On, 1 = Off
LED[5]: LED D23 – 0 = On, 1 = Off
LED[6]: LED D24 – 0 = On, 1 = Off
LED[7]: LED D25 – 0 = On, 1 = Off
7SEG[0]: D36 Segment a – 0 = On, 1 = Off
7SEG[1]: D36 Segment b – 0 = On, 1 = Off
7SEG[2]: D36 Segment c – 0 = On, 1 = Off
7SEG[3]: D36 Segment d – 0 = On, 1 = Off
7SEG[4]: D36 Segment e – 0 = On, 1 = Off
7SEG[5]: D36 Segment f – 0 = On, 1 = Off
7SEG[6]: D36 Segment g – 0 = On, 1 = Off
7SEG[7]: D36 Segment dp – 0 = On, 1 = Off
PMOD2[0]: J8 Pin 1 I/O
PMOD2[1]: J8 Pin 2 I/O
PMOD2[2]: J8 Pin 3 I/O
PMOD2[3]: J8 Pin 4 I/O
PMOD2[4]: J8 Pin 7 I/O
PMOD2[5]: J8 Pin 8 I/O
PMOD2[6]: J8 Pin 9 I/O
PMOD2[7]: J8 Pin 10 I/O
PMOD2DIR[0]: J8 Pin 1 Direction – 0 = Input, 1 = Output
PMOD2DIR[1]: J8 Pin 2 Direction – 0 = Input, 1 = Output
PMOD2DIR[2]: J8 Pin 3 Direction – 0 = Input, 1 = Output
PMOD2DIR[3]: J8 Pin 4 Direction – 0 = Input, 1 = Output
PMOD2DIR[4]: J8 Pin 7 Direction – 0 = Input, 1 = Output
PMOD2DIR[5]: J8 Pin 8 Direction – 0 = Input, 1 = Output
PMOD2DIR[6]: J8 Pin 9 Direction – 0 = Input, 1 = Output
PMOD2DIR[7]: J8 Pin 10 Direction – 0 = Input, 1 = Output

Note: Register function is not supported in the initial release.

2.12. SPI Manager IP Design Details

The Serial Peripheral Interface (SPI) is a high-speed synchronous, serial, full-duplex interface that allows a serial bitstream of configured length (8, 16, 24, and 32 bits) to be shifted into and out of the device at a programmed bit transfer rate. The Lattice SPI Manager IP Core is normally used to communicate with external SPI subordinate devices such as display drivers, SPI EPROMS, and analog-to-digital converters.

The SPI Manager IP is used to be integrated in node system SOC design as defined in node system top level architectural diagram. This IP can be controlled by C/C++ APIs of node system CPU to read/write data from/to certain SPI based peripheral/sensor. These C/C++ based APIs can be controlled by main system as well.

This section only provides minimum details on the SPI Manager IP required for integration and controlling. For more details, refer to SPI Controller IP Core –User Guide (FPGA-IPUG-02069).

2.12.1. Overview

The SPI Manager IP Core allows the CPU inside the FPGA to communicate with multiple external SPI subordinate devices. The data size of the SPI transaction can be configured to be 8, 16, 24, or 32 bits. This IP is designed to use an internal FIFO of configurable depth to minimize the host intervention during data transfer. SPI Manager IP Core supports all SPI clocking modes – combinations of Clock Polarity (CPOL) and Clock Phase (CPHA) to match the settings of external devices.

The SPI Manager IP provides a bridge between LMMI/AHB-Lite/APB and standard external SPI bus interfaces (functional diagram is shown in Figure 2.15. On the external, off-chip side the SPI Manager Controller IP has a standard SPI bus interface. On the internal, on-chip side, the SPI Manager Controller IP has LMMI/AHB-Lite/APB subordinate interface depending on the Interface attribute settings.

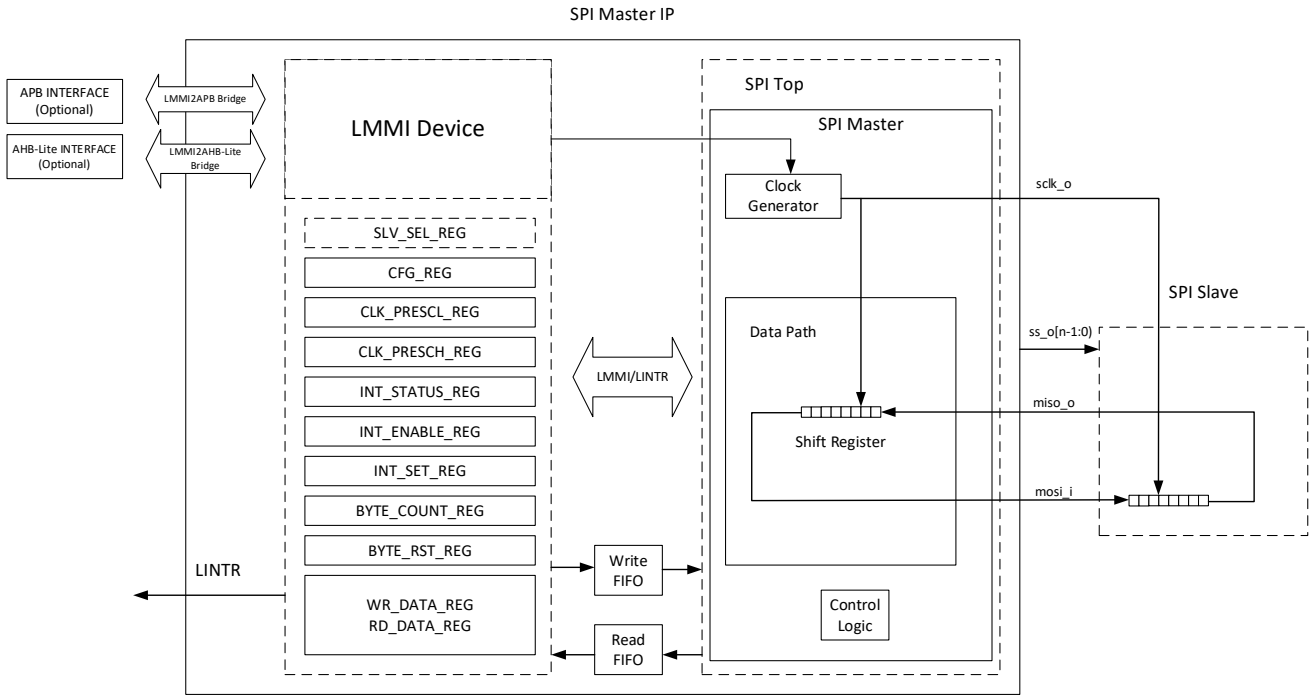

Figure 2.15. SPI Manager IP Core Block Diagram

2.12.2. SPI Manager Register Map

Table 2.45. SPI Manager Register Map

Offset LMMI	Offset APB/AHBL	Register Name	Access Type	Description
0x0	0x00	WR_DATA_REG	WO	Write Data Register
0x0	0x00	RD_DATA_REG	RO	Read Data Register
0x1	0x04	SLV_SEL_REG	RW	Subordinate Select Register
0x2	0x08	CFG_REG	RW	Configuration Register
0x3	0x0C	CLK_PRESCL_REG	RW	Clock Pre-Scaler Low Register
0x4	0x10	CLK_PRESCH_REG	RW	Clock Pre-Scaler High Register
0x5	0x14	INT_STATUS_REG	RW1C	Interrupt Status Register
0x6	0x18	INT_ENABLE_REG	RW	Interrupt Enable Register
0x7	0x1C	INT_SET_REG	WO	Interrupt Set Register
0x8	0x20	WORD_CNT_REG	RO	Word Count Register
0x9	0x24	WORD_CNT_RST_REG	WO	Word Count Reset Register
0xA	0x28	TGT_WORD_CNT_REG	RW	Target Word Count Register
0xB	0x2C	FIFO_RST_REG	WO	FIFO Reset Register

Offset LMMI	Offset APB/AHBL	Register Name	Access Type	Description
0xC	0x30	SLV_SEL_POL_REG	RW	Subordinate Select Polarity Register
0xD	0x34	FIFO_STATUS_REG	RO	FIFO Status Register
0xE 0xF	0x38-0x3C	Reserved	RSVD	Reserved. Write access is ignored and 0 is returned on read access.

Table 2.45 lists the address map and specifies the registers available to the user. The offset of each register is dependent on the Interface attribute setting as follows:

- Interface selected to be LMMI: the offset increments by one.
- Interface selected to be either AHBL or APB: the offset increments by four to allow easy interfacing with the Processor and System Buses. In this mode, each register is 32-bit wide wherein the upper unused bits are reserved, and the lower bits are described in each register description.

Notes:

1. For more details on the registers above, refer to the SPI Controller IP Core –User Guide (FPGA-IPUG-02069).
2. The RD_DATA_REG and WR_DATA_REG share the same offset. Write access to this offset goes to WR_DATA_REG while read access goes to RD_DATA_REG.

2.12.3. Programming Flow

2.12.3.1. Initialization

The following SPI Manager registers should be set properly before performing SPI transaction:

- SLV_SEL_REG – Set 1'b1 to the bit for the target node. Set 1'b0 to other bits.
- SLV_SEL_POL_REG – may be configured once after reset since this setting is usually fixed.
- CLK_PRESCL_REG – Set based on target sclk_o frequency.
- CLK_PRESCH_REG – Set based on target sclk_o frequency.

The CPU needs to update the above registers only when SPI Manager aster is switching to different subordinate device. This means there is no need to perform initialization again if the next transaction is for the currently selected subordinate device.

2.12.3.2. Transmit/Receive Operation

The following are the recommended steps on performing the SPI transaction. This assumes that the module is not currently performing any operation.

1. Set the following CFG_REG fields according to the target Subordinate settings: cpha, cpol, ssnp and lsb_first. Set the only_write field based on the current transaction. If CFG_REG.only_write is 1'b0, SPI manager performs both transmit and receive operations (full-duplex). On the other hand, if CFG_REG.only_write is 1'b1, SPI Manager IP Core performs Transmit operation only.
2. Set TGT_WORD_CNT_REG according to the number of words to transfer.
3. Reset WORD_CNT_REG by writing 8'hFF to Word Count Reset Register
4. Write data words to WR_DATA_REG, amounting to ≤ FIFO Depth.
5. Optional: If interrupt mode is desired, enable target interrupts in INT_ENABLE_REG If number of words to transfer is ≤ FIFO Depth, set tr_cmp_en = 1'b1. If number of words to transfer is > FIFO Depth, set the following: tx_fifo_aempty_en = 1'b1 and tr_cmp_en = 1'b1. Other interrupts not specified above are disabled.
6. If total number of words to transfer > FIFO Depth, wait for Transmit FIFO Almost Empty Interrupt.
   a. If polling mode is desired, read INT_STATUS_REG until tx_fifo_aempty_int asserts.
   b. If interrupt mode is desired, simply wait for interrupt signal to assert, then read INT_STATUS_REG and check that tx_fifo_aempt_int is asserted.
7. Clear Transmit FIFO Almost Empty Interrupt by writing 1'b1 to INT_STATUS_REG.tx_fifo_aempty_int. Clearing all interrupts by writing 8'hFF to INT_STATUS_REG is also okay since the user is not interested in other interrupts for this recommended sequence.

8. Write data words to WR_DATA_REG, amounting to less than or equal to (FIFO Depth – TX FIFO Almost Empty Flag).
9. If CFG_REG.only_write = 1'b0, read all the data in RD_DATA_REG. It is expected that Receive FIFO has (FIFO Depth – TX FIFO Almost Empty Flag - 1) amount of data words. Read INT_STATUS_REG.rx_fifo_ready_int to check if RD_DATA_REG is already empty.
10. If there is remaining data to transfer, go back to Step 6. Note that you can read Word Count Register to determine the number of words already transferred in SPI interface.
11. Wait for Transfer Complete Interrupt.

    If polling mode is desired, read INT_STATUS_REG until tr_cmp_int asserts.

    If interrupt mode is desired, set INT_ENABLE_REG = 8'h80 then wait for interrupt signal to assert. Then read INT_STATUS_REG and check that tr_cmp_int is asserted.
12. Clear all interrupts by writing 8'hFF to INT_STATUS_REG.
13. If CFG_REG.ONLY_WRITE = 1'b0, read all the data in RD_DATA_REG. Read INT_STATUS_REG.rx_fifo_ready_int to check if RD_DATA_REG is already empty

a.
b.

2.13. I²C Manager IP Design Details

The I²C (Inter-Integrated Circuit) bus is a simple, low-bandwidth, short-distance protocol. It is often seen in systems with peripheral devices that are accessed intermittently. It is commonly used in short-distance systems, where the number of traces on the board should be minimized. The device that initiates the transmission on the I²C bus is commonly known as the Manager, while the device being addressed is called the Subordinates.

The I²C Manager IP is used to be integrated in node system SOC design as defined in node system top level architectural diagram. This IP can be controlled by C/C++ APIs of node system CPU to read/write data from/to certain I²C based peripheral/sensor. These C/C++ based APIs can be controlled by main system as well.

This section only provides minimum details of the I²C Manager IP required for the integration and controlling. Refer to the I²C Manager IP user guide for more details.

2.13.1. Overview

The I²C Manager IP Core accepts commands from LMMI/APB interface through the register programming. These commands are decoded into I²C read/write transactions to the external I²C subordinate device. The I²C bus transactions can be configured to be 1 to 256 bytes in length.

The I²C Manager Controller can operate in interrupt or polling mode. This means that the CPU can choose to poll the I²C Manager for a change in status at periodic intervals (Polling Mode) or wait to be interrupted by the I²C Manager Controller when data needs to be read or written (Interrupt Mode).

Figure 2.16 shows the functional diagram of the I²C Manager Controller.

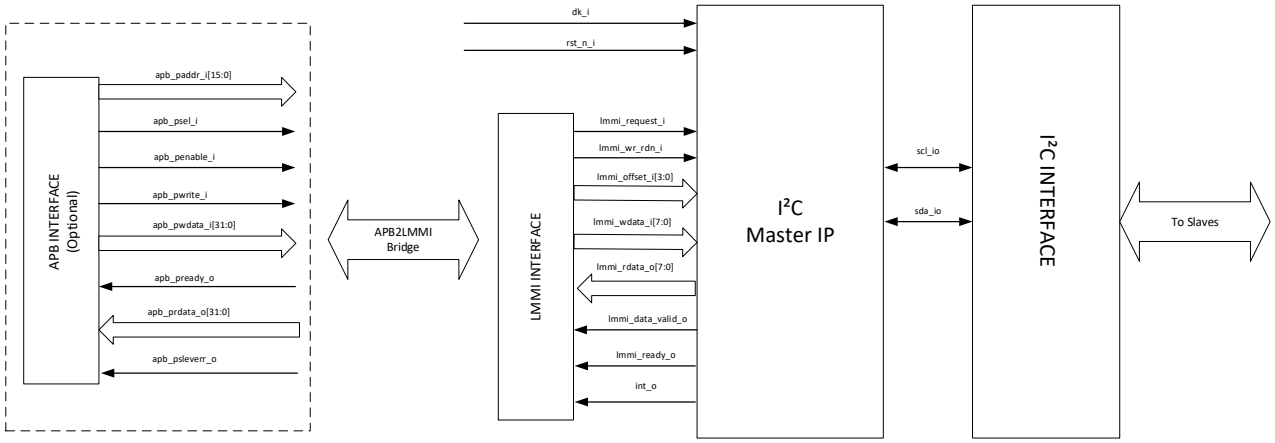

Figure 2.16. I²C Manager Controller IP Core Functional Diagram

2.13.2. I²C Manager Register Map

The CPU can control the I²C Manager IP Core by writing to and reading from the configuration registers. The I²C Manager IP Core configuration registers can be performed at the run-time.

Table 2.46 lists the address map and specifies the registers available to you. The offset of each register is dependent on attribute APB Mode Enable setting as follows:

- APB Mode Enable is Unchecked – the offset increments by 1.
- APB Mode Enable is Checked – the offset increments by 4 to allow easy interfacing with the Processor and System Buses. In this mode, each register is 32-bit wide wherein the upper bits [31:8] are reserved and the lower 8 bits [7:0] are described in the Programming Flow section.

The RD_DATA_REG and WR_DATA_REG share the same offset. Write access to this offset goes to WR_DATA_REG while read access goes to RD_DATA_REG.

Table 2.46. I²C Manager IP Core Registers Summary

Offset LMMI	Offset APB/AHBL	Register Name	Access Type	Description
0x0	0x00	WR_DATA_REG	WO	Write Data Register
0x0	0x00	RD_DATA_REG	RO	Read Data Register
0x1	0x04	SLAVE_ADDRL_REG	RW	Subordinate Address Lower Register
0x2	0x08	SLAVE_ADDRH_REG	RW	Subordinate Address Higher Register
0x3	0x0C	CONTROL_REG	WO	Control Register
0x4	0x10	TGT_BYTE_CNT_REG	RW	Byte Count Register
0x5	0x14	MODE_REG	RW	Mode Register
0x6	0x18	CLK_PRESCL_REG	RW	Clock Prescaler Low Register
0x7	0x1C	INT_STATUS1_REG	RW1C	First Interrupt Status Register
0x8	0x20	INT_ENABLE1_REG	RO	First Interrupt Enable Register
0x9	0x24	INT_SET1_REG	WO	First Interrupt Set Register
0xA	0x28	INT_STATUS2_REG	RW1C	Second Interrupt Status Register
0xB	0x2C	INT_ENABLE2_REG	RO	Second Interrupt Enable Register
0xC	0x30	INT_SET2_REG	WO	Second Interrupt Set Register
0xD	0x34	FIFO_STATUS_REG	RO	FIFO Status Register
0xE	0x38	SCL_TIMEOUT_REG	RW	SCL Timeout Register
0xF	0x3C	Reserved	RSVD	Reserved. Write access is ignored and 0 is returned on read access.

Note: RW1C (Writing 1'b1 on register bit clears the bit to 1'b0. Writing 1'b0 on register bit is ignored). For more details on the registers above, refer to I²C Manager IP Core – Lattice Radiant Software User Guide (FPGA-IPUG-02071).

2.13.3. Programming Flow

2.13.3.1. Initialization

The following I²C Manager registers can be set outside of the actual transaction sequence. These should be set properly before starting an I2C transaction:

- SLAVE_ADDRL_REG, SLAVE_ADDRH_REG – Set the address of the target Subordinate Device
- CLK_PRESCL_REG – Set based on target scl_io frequency. The upper bits, MODE_REG and clk_presc_high, are set during transaction because they are grouped with mode register.
- SCL_TIMEOUT_REG – Set to 8'h00 if the user does not want to check the SCL timeout or set to desired timeout value.
- INT_ENABLE2_REG – it is recommended to enable all interrupts in this register to check for error/unexpected event.

When accessing multiple devices, the SLAVE_ADDRL_REG or SLAVE_ADDRH_REG registers should be set prior to transaction.

2.13.3.2. Writing to the Subordinate Device

The following are the recommended steps for performing I²C write transaction, this assumes that the module is not currently performing any operation and initialization is completed.

To perform I²C write transaction:

1. Set the following MODE_REG fields according to the desired transfer mode: bus_speed_mode, addr_mode, ack_mode, clk_presc_high. Set the trx_mode field to 1'b0 for write transaction.
2. Set TGT_BYTE_CNT_REG according to the number of bytes to transfer.
3. Write data to WR_DATA_REG, amounting to ≤ FIFO Depth.
4. Set CONTROL_REG.start to 1'b1 to start the I2C transaction.

   Optional: If interrupt mode is desired, Enable target interrupts in INT_ENABLE1_REG. If number of words to transfer is ≤ FIFO Depth, set tr_cmp_en = 1'b1 If number of words to transfer is > FIFO Depth, set the following: tx_fifo_aempty_en = 1'b1 and tr_cmp_en = 1'b1. Other interrupts in this register are disabled.
5. If total number of bytes to transfer > FIFO Depth, wait for Transmit FIFO Almost Empty Interrupt. If polling mode is desired, read INT_STATUS1_REG until tx_fifo_aempty_int asserts. If interrupt mode is desired, simply wait for interrupt signal to assert, then read INT_STATUS1_REG and check that tx_fifo_aempt_int is asserted. In both cases, read also INT_STATUS2_REG to ensure that the transfer is good. I²C Manager IP Core
6. Clear Transmit Buffer Almost Empty Interrupt by writing 1'b1 to INT_STATUS1_REG.tx_fifo_aempty_int. Clearing all interrupts in this register by writing 8'hFF to INT_STATUS1_REG is also okay since the user is not interested in other interrupts for this recommended sequence.
7. Write data to WR_DATA_REG, amounting to less than or equal to (FIFO Depth – TX FIFO Almost Empty Flag).
8. If there is remaining data to transfer, go back to Step 6.
9. Wait for Transfer Complete Interrupt.
10. If polling mode is desired, read INT_STATUS1_REG until tr_cmp_int asserts. If interrupt mode is desired, set INT_ENABLE1_REG = 8'h80 then wait for interrupt signal to assert. Read INT_STATUS1_REG and if tr_cmp_int is asserted.
11. Clear all interrupts by writing 8'hFF to INT_STATUS1_REG.

2.13.3.3. Reading from the Subordinate Device

The following are the recommended steps for performing I2C read transaction, assuming that the module is currently not performing any operation and if initialization is completed.

To perform I²C read transaction:

1. Set the following MODE_REG fields according to the desired transfer mode: bus_speed_mode, addr_mode, ack_mode, clk_presc_high. Set the trx_mode field to 1'b1 for read transaction.
2. Set TGT_BYTE_CNT_REG according to the number of bytes to transfer.
3. Set CONTROL_REG.start to 1'b1 to start the I2C transaction.

   Optional: If interrupt mode is desired, enable target interrupts in INT_ENABLE1_REG If number of words to transfer is ≤ FIFO Depth, set tr_cmp_en = 1'b1.
4. If number of words to transfer is > FIFO Depth, set the following: rx_fifo_afull_en = 1'b1 and tr_cmp_en = 1'b1. Other interrupts in this register are disabled.
5. If total number of bytes to receive > FIFO Depth, wait for Receive FIFO Almost Full Interrupt. If polling mode is desired, read INT_STATUS1_REG until rx_fifo_afull_int asserts. If interrupt mode is desired, wait for the interrupt signal to assert, and then read INT_STATUS1_REG and check if rx_fifo_afull_int is asserted. In both cases, read also INT_STATUS2_REG to ensure that the transfer is good.
6. Clear Receive FIFO Almost Full Interrupt by writing 1'b1 to INT_STATUS1_REG.rx_fifo_afull_int. Clearing all interrupts in this register by writing 8'hFF to INT_STATUS1_REG is also okay since the user is not interested in other interrupts for this recommended sequence.
7. Read all data from RD_DATA_REG. It is expected the amount of received data is less than or equal to (FIFO Depth – TX FIFO Almost Empty Flag). Read FIFO_STATUS_REG to confirm if Receive FIFO is emptied.
8. If there is remaining data to receive, go back to Step 5.

9. Wait for Transfer Complete Interrupt. If polling mode is desired, read INT_STATUS1_REG until tr_cmp_int asserts If interrupt mode is desired, set INT_ENABLE1_REG = 8'h80 and wait for the interrupt signal to assert. Read INT_STATUS1_REG and check that tr_cmp_int is asserted.
10. Clear all interrupts by writing 8'hFF to INT_STATUS1_REG.
11. Read all the remaining data from RD_DATA_REG.

2.14. UART IP Design

The Lattice Semiconductor UART (Universal Asynchronous Receiver/Transmitter) IP Core is designed for use in serial communication, supporting the RS-232.
The UART IP is used to be integrated in the node system SOC design as defined in node system top level architectural diagram. This IP can be controlled by C/C++ APIs of node system CPU to read/write data from/to certain UART/modbus based peripheral/sensor. These C/C++ based APIs can be controlled by main system as well.
This sections only provides minimum details of the UART IP required for the integration and controlling. Refer to the UART IP user guide for more details.

2.14.1. Overview

The UART IP Core performs two main functions:

- Serial-to-parallel conversion on data characters received from an external UART device.
- Parallel-to-serial conversion on data characters received from the Host located in the FPGA.

The CPU can read the complete status of the UART at any time during the functional operation. Status information reported includes the type and condition of the transfer operations being performed by the UART IP Core, as well as any error conditions (parity, overrun, framing, or break interrupt).

The UART IP has implemented a processor-interrupt system similar to UART 16450. Interrupts can be programmed to your requirements, minimizing the computing required to handle the communications link. The UART IP currently does not implement the MODEM-control feature of UART 16450.

The registers of UART IP Core are accessed by the CPU (FPGA internal components) through an AMBA APB interface. The functional block diagram of UART IP Core is shown in Figure 2.17. The dashed lines in the figure are optional components/signals, which means they may not be available in the IP when disabled in the attribute.

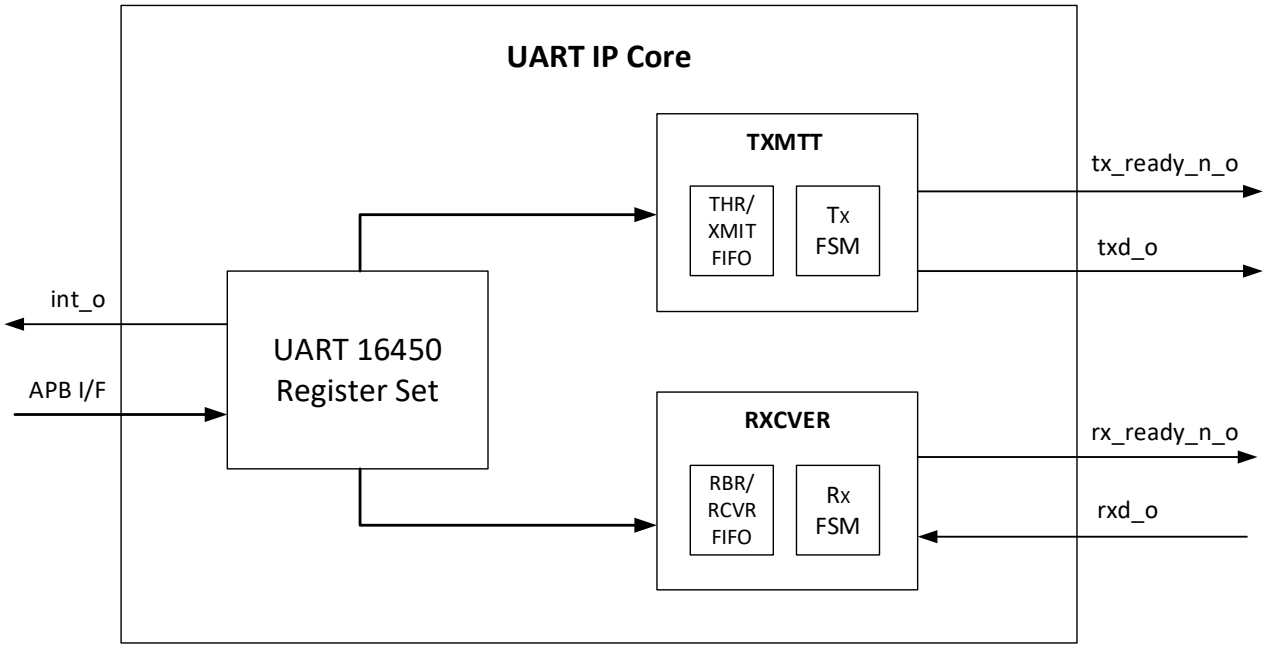

Figure 2.17. UART IP Core Functional Block Diagram

2.14.1.1. UART Register Description

The register address map, shown in Table 2.47, specifies the available IP Core registers. This is based on register set of UART 16450, but the offset address is changed to simplify the access to each register. The offset of each register increments by four to allow easy interfacing with the Processor and System Buses. In this case, each register is 32-bit wide wherein the lower 8 bits are used, and the upper 24 bits are unused. The unused bits are treated as reserved – write access is ignored and read access returns 0.

Table 2.47. UART Register Map

Offset	Register Name	Access Type	Description
0x00	RBR	RO	Receive Buffer Register
0x00	THR	WO	Transmitter Holding Register
0x04	IER	RW	Interrupt Enable Register
0x08	IIR	RO	Interrupt Identification Register
0x0C	LCR	RW	Line Control Register
0x10	Reserved	RSVD	Reserved
0x14	LSR	RO	Line Status Register
0x18-0x1C	Reserved	RSVD	Reserved
0x20	DLR_LSB	WO	Divisor Latch Register LSB
0x24	DLR_MSB	WO	Divisor Latch Register MSB
0x28-0x3C	Reserved	RSVD	Reserved

Note: Details of Registers is given in UART IP Core – Lattice Propel Builder User Guide (FPGA-IPUG-02105).

2.14.2. Programming Flow

2.14.2.1. Initialization

The following UART register fields should be set properly before performing UART transaction:

- Line Control Register – even_parity_sel, parity_en, stop_bit_ctrl, char_len_sel
- Divisor Latch Registers – divisor_msb, divisor_lsb

These should match the corresponding setting in the communicating UART for the serial transaction to be successful.

Note that reset values of these register fields are configurable during IP generation. Thus, in some applications, initialization step is not necessary when attributes are properly set.

2.14.2.2. Transmit Operation

The following are the steps for transmitting character data through the UART IP Core. This is assuming that the IP is not performing transmit operation or at least the XMIT FIFO is empty.

Transmit Operation – Interrupt Mode

To perform transmit operation in interrupt mode:

1. Write the data to THR. In FIFO mode, user can write up to 16-character data.
2. Set IER.thre_int_en=1'b1 to enable Transmit Holding Register Empty interrupt.
3. Wait for Transmit Holding Register Empty interrupt to assert.
4. Wait for interrupt assertion and check that IIR[3:0]= 4'b0010.
5. If the user needs to send more characters, repeat Steps 1-3 until all characters are sent.
6. When using interrupt, set IER.thre_int_en=1'b0 to disable the interrupt.

Transmit Operation – Polling Mode

To perform transmit operation in polling mode:

1. Write a data to THR. It is recommended not to enable FIFO for polling mode to save resource.
2. Read LSR until the thr_empty bit asserts.
3. If you need to send more characters, repeat steps 1 and 2 until all characters are sent.

2.14.2.3. Receive Operation

The following are the steps for the receiving character data through the UART IP Core. This is assuming that the IP core is not performing receive operation.

Receive Operation – Interrupt Mode

To perform receive operation in interrupt mode:

1. Enable the following interrupts:
   - Received Data Available Interrupt (IER.rda_int_en=1'b1) – to notify the host that a data is received.
   - Receiver Line Status interrupt (IER.rls_int_en=1'b1) – to notify the host of receive status such as error and break condition.
2. Wait for interrupt assertion and check that IIR[2:0]= 3'b100 (Receive Data Available). If Receiver Line Status Interrupt asserts (IIR[2:0]=3'b110), read the LSR to determine the cause.
   a.
   b.
3. If Receiver Line Status Interrupt does not occur, read the character data from RBR:
   - If Receive Data Available Interrupt occurs, read a data from RBR.
   - If Character Timeout Interrupt occurs, read LSR. If LSR.data_rdy=1'b1, read RBR.
4. Repeat steps 2-3 until all expected data are received.
   a.
   b.

Receive Operation – Polling Mode

To perform receive operation in polling mode:

1. Read LSR until the thr_empty bit asserts. Also, check that no error status bits are asserted.
2. Read RBR if there is no error.
3. If the user needs to receive more characters, repeat Steps 1 and 2 until all characters are received.

2.14.2.4. Data Format

The character data written to THR and read from RBR is in little endian format as shown in Figure 2.18.

7-Bit Data

0	D6	D5	D4	D3	D2	D1	D0

8-Bit Data

D7	D6	D5	D4	D3	D2	D1	D0

Figure 2.18. UART Data Format

2.15. TSE MAC

Tri-Speed Ethernet Media Access Controller (TSEMAC) IP core is a complex core containing all necessary logic, interfacing, and clocking infrastructure necessary to integrate an external industry-standard Ethernet PHY with an internal processor efficiently and with minimal overhead.

The TSEMAC IP core supports the ability to transmit and receive data between the standard interfaces, such as APB or AHB-Lite, and an Ethernet network. The main function of TSEMAC IP is to ensure that the Media Access rules specified in the 802.3 IEEE standard are met while transmitting a frame of data over Ethernet. On the receiving side, the TSEMAC extracts different components of a frame and transfers them to higher applications through the FIFO interface.

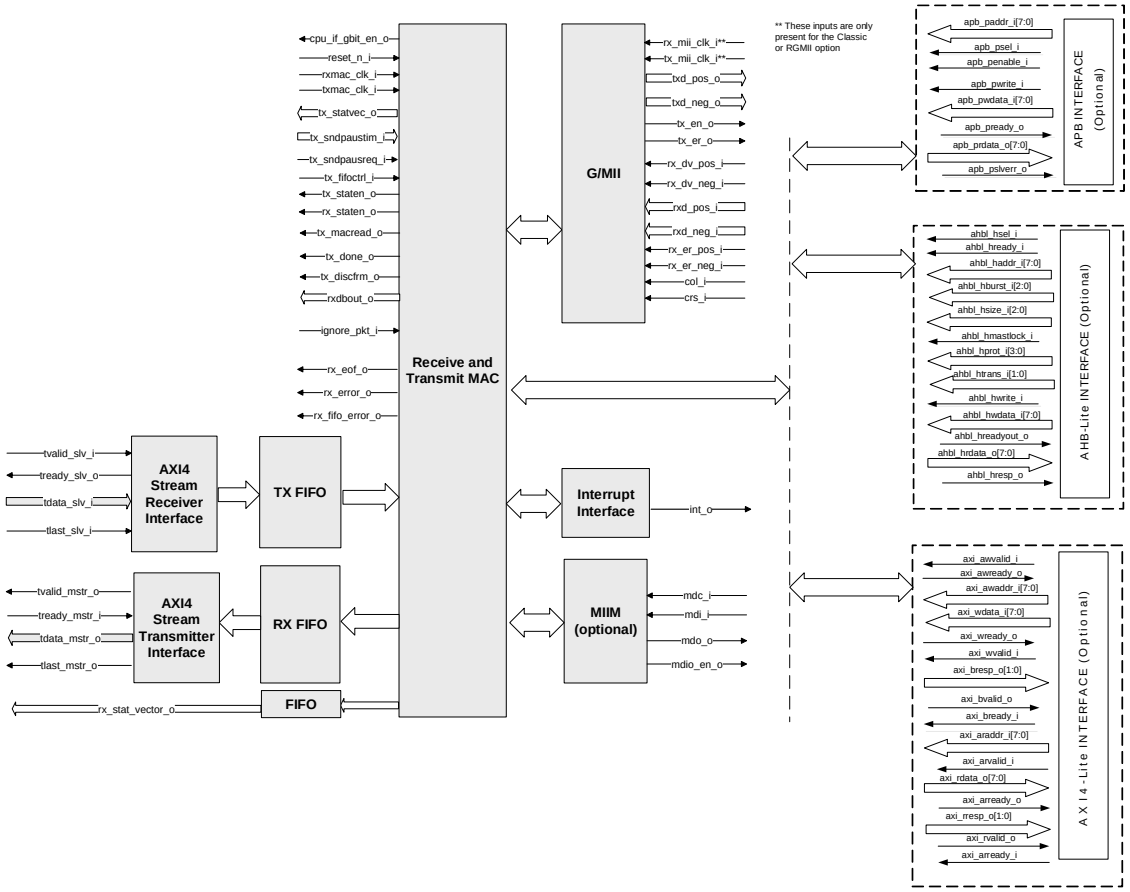

Figure 2.19. Classic TSEMAC IP Top-Level Block Diagram

The register map is shown in section 2.4 of TSE MAC IPUG. For more details, refer to Tri-Speed Ethernet MAC IP Core User Guide (FPGA-IPUG-02084).

2.16. SGMII IP Design

The Serial Gigabit Media Independent Interface (SGMII) connects Ethernet Media Access Controllers (MAC) and Physical Layer Devices (PHY). This IP core may be used in bridging applications and/or PHY implementations. SGMII/Gb Ethernet PCS IP core converts GMII frames into 8-bit code groups in both transmit and receive directions and performs auto-negotiation with a link partner as described in the Cisco SGMII and IEEE 802.3 specifications. SGMII IP is a connection bus for MAC and PHY and is often used in bridging applications and/or PHY implementations. It is particularly widely used as an interface for a discrete Ethernet PHY chip.

The on board reset and the DDR Initialization signals are ANDed and given as the reset of the IP in the current design to ensure that any data transfer happens only after the DDR is initialized.

The IP settings are shown in Figure 2.20.

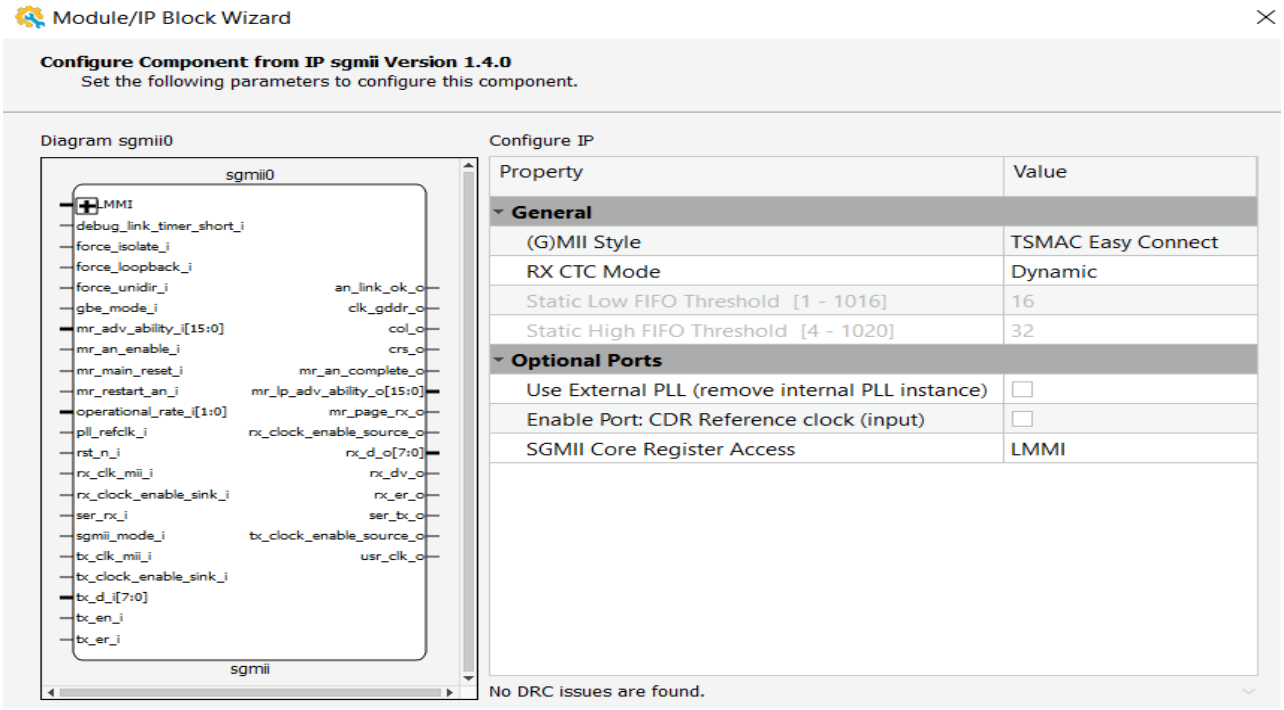

Figure 2.20. SGMII IP Settings

For more details, refer to SGMII and Gb-Ethernet PCS IP Core User Guide (FPGA-IPUG-02077).

2.17. FPGA Config Module Design

Nexus configuration logic provides an LMMI interface to allow user logic residing inside the FPGA fabric to access the device configuration (CFG) functionalities. To achieve this, the user has to instantiate the CONFIG_LMMIA primitive in the design.To make sure the clock for the configuration engine is enabled, the user has to instantiate the OSC primitive in the design as well. The FPGA config module will be designed to execute LSC_REFRESH command which is equivalent to toggling PROGRAMN pin to automatically switch to alternate pattern (golden pattern).
The module will contain connections of the two primitives as shown below.

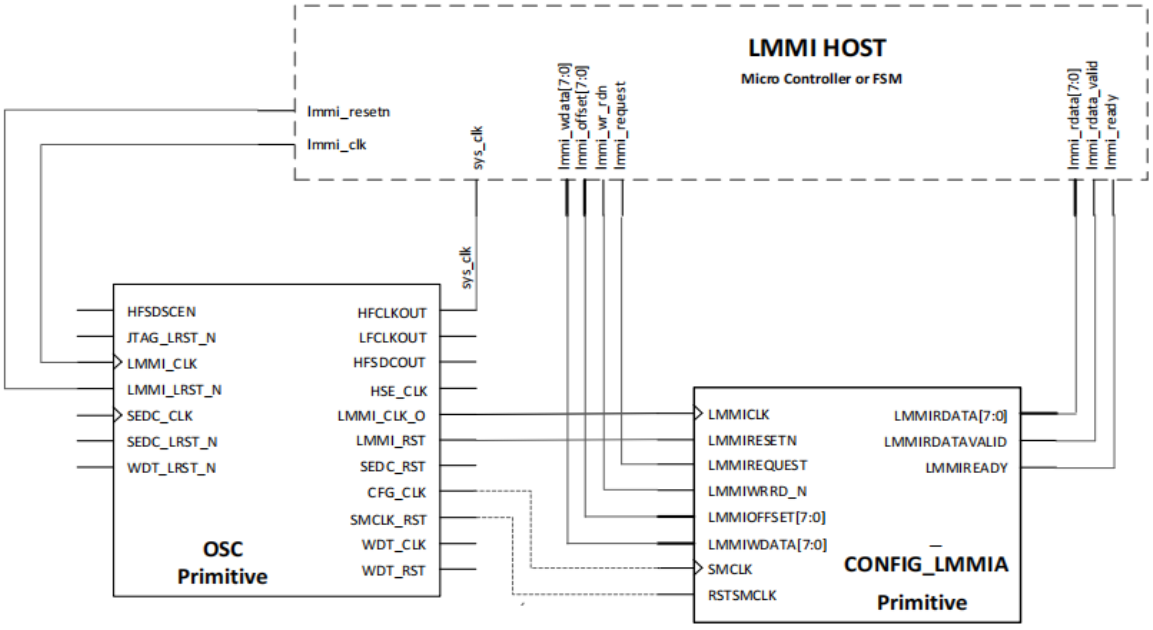

Figure 2.21. CONFIG_LMMIA Primitive and OSC Primitive Connection

The module also has the LMMI driver logic, as shown in Figure 2.22, to execute the LSC_REFRESH command. The LMMI interface runs on the sys_clk of 28.153 MHz.

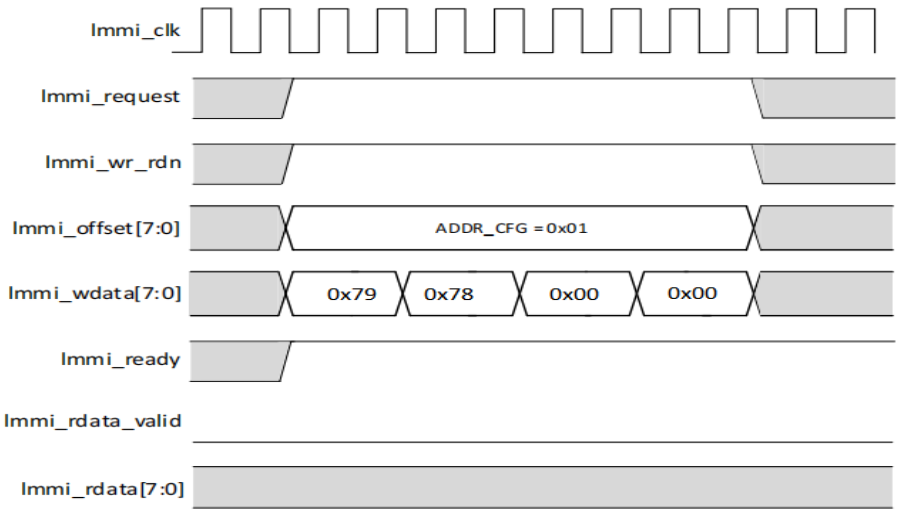

Figure 2.22. LMMI LSC_REFRESH Command Execution

2.18. SFP Config Design Details

This module contains the logic for SFP configuration. It has LMMI module and the I²C master module. The I²C master derives the LMMI to do the SFP configuration. It uses the link ok signal of the SGMII IP and SFP Absent signal, which generates the SFP Disable signal.

The IP user interface is shown in Figure 2.23.

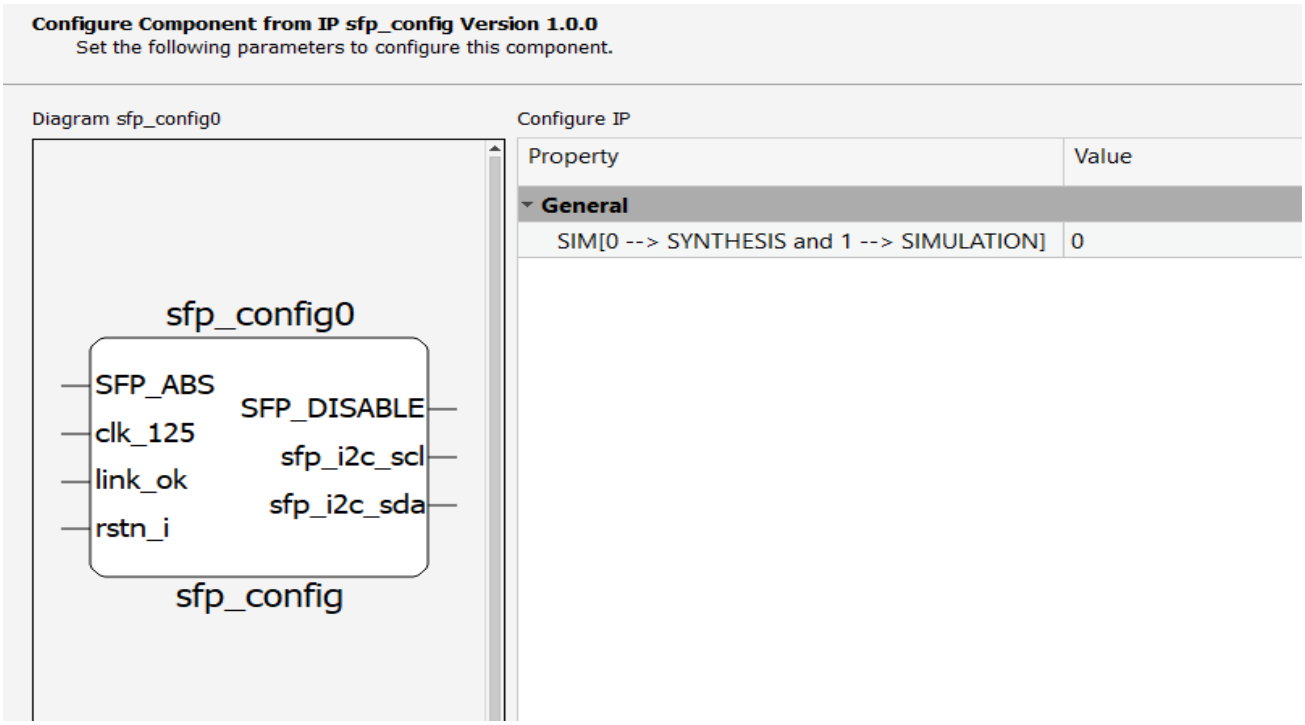

Figure 2.23. SFP Config User Interface

3. Resource Utilization

The resource utilization for the Main System is shown in Table 3.1.

Table 3.1. Main System Resource Utilization

Blocks	LUTs	EBRs	LRAMs	DSPs	Comments
RISC-V CPU	5757	18	—	6	—
ISR RAM	116	32	—	—	—
Data RAM (System Memory)	884	—	2	—	—
AXI4 Interconnect	17719	—	—	—	—
APB Interconnect	78	—	—	—	—
FIFO DMA	924	16	—	—	—
EtherConnect	4810	16	—	—	—
UART	271	—	—	—	—
GPIO	108	—	—	—	—
PLL	1	—	—	—	—
SPI Flash Controller	508	1	—	—	—
AXI2APB	269	0	—	—	—
CNN Coprocessor Unit (CCU)	992	—	—	4	—
Reset Sync	78	—	—	—	—
Multiport Extension	16414	—	—	65	—
UDP Stack	8870	—	—	4	—
SGMII MAC Wrapper	4346	7	—	—	—
Top-level	2	—	—	—	—
Total	**62147**	**90**	**2**	**79**	**—**

The resource utilization for the Node System is shown in Table 3.2.

Table 3.2. Node System Resource Utilization

Blocks	LUTs	EBRs	LRAMs	DSP MULT	Comments
RISC-V CPU	2537	2	—	—	—
ISR RAM	51	16	—	—	—
Data RAM (System Memory)	155	0	2	—	—
AHBL Interconnect	1721	—	—	—	—
APB Interconnect	14	—	—	—	—
FIFO DMA	754	16	—	—	—
EtherConnect	4209	11	—	—	—
SPI Flash Controller	229	1	—	—	—
AHBL2APB	148	—	—	—	—
Motor Control Data Collector	4152	17	—	15.5	—
UART	261	—	—	—	—
I^2C Manager	585	—	—	—	—
SPI Manager	398	—	—	—	—
Top-level	2	—	—	—	—
Total	**15216**	**63**	**2**	**15.5**	—

4. Software APIs

4.1. Main System APIs

4.1.1. Tasks of the Main System

The Main System acts as an interface between the user interface and the node-system, which controls the motor IP. The commands are then sent to the nodes for configuration through EtherConnect. The Main System also enables the user interface to monitor various parameters of the motors. The system also receives commands from the GPIO switches attached on the board and sends these commands to the nodes for configuration through EtherConnect as well.

The tasks to be carried out by the Main System can be categorized as follows:

- System Initialization
  This API is used to configure the EtherConnect and establish communication between the Main system and nodes. This takes place as soon as there is a power cycle or reset is pressed.
- Handle all the interrupts (GPIO, EtherConnect) and respond to the interrupts by taking appropriate actions.
- Communication with the host system, Node System, and mechanical switches occur through interrupts and the Main System takes appropriate actions based on the interrupts caused. The priority order of all the interrupts is GPIO > EtherConnect.
- Switch Configuration over GPIO
  You can start, stop, accelerate, and decelerate the motors with the help of switches provided. The Main System configures the node motor IP as per the switch configuration.
- Communicate with host system user interface over Ethernet
  The host system user interface sends configuration data and status check commands to the Main System, and the Main System responds based on the command.
- Communicate with Node System and motor IP over EtherConnect
  As per the commands received by the Main System, it creates burst packets to send to the Node System, that the Node System then receives and implements them. This communication between the main and Node System happens over EtherConnect and at a given time, a maximum of 256 bytes can only be transmitted from either direction.
- ISR3_EtherConnect
  *static void etherConnect_isr (void *ctx)*
  The primary function of the EtherConnect ISR function is to set the interrupt flag, acknowledge the interrupt, and return a value. The EtherConnect interrupt is used as an acknowledgement of the completion of a single transaction of a command sent by the Main System to the Node System. The IRQ value for EtherConnect is IRQ3.
- System Initialization API
  int system_initialisation (void)
  This API is present in the main.c file. It does not take any parameter and returns an integer value. It returns 0 if everything is successfully completed or a – 1 if there is an error.
  This API is used to establish communication between the Main System and the Node System. It enables the DMA FIFO module and sends 10 broadcast packets to detect the number of nodes available and active in the whole setup. By reading the PHY Link Status register, it affirms whether the communication is established or not, and accordingly, turns ON the Main System LEDs. This API then sends three training packets and one normal packet to the Node System through the EtherConnect to affirm the connection establishment with the Node System.
- Motor Configuration API
  int motor_config_api(uint32_t address, uint32_t data, uint32_t multi)

  This API is present in the main.c file. It needs three parameters namely:
  - address: signifies a register in the Motor Control IP
  - data: what needs to be written in that register
  - multi: data to be transmitted on multiple chains or selected chain only

It returns the following integer values:

- 0: if everything is correct
- –1: if there was any error

The API is called when there is a requirement to configure a register in the Motor Control IP of the Node System. This occurs in two cases:

- when there is an ON switch on any GPIO

The API creates burst packets which are sent to the Node System over EtherConnect. The header in the burst packet indicates that a particular packet is for Motor Configuration and for which nodes this packet is intended. Once the burst packet is written in a FIFO module, it is sent to the Node System by a trigger of 1 to 0 signal in a Start Transaction Register. After the Node System completes the task successfully, the Main System receives an interrupt and validates the value of the interrupt info register. Upon the confirmation of the value of the interrupt info register, this API returns a 0 value or a –1 if there is an error.

- Motor Status API

  int motor_status_api(uint32_t address, uint32_t multi)

  This API is present in the main.c file. It needs one parameter:

  - address: signifies a register in the Motor Control IP
  - multi: EtherConnect packet to be transmitted on multiple chains or selected chains only

  It returns the following integer values:

  - 0: if all tasks are successfully completed
  - –1: if there is an error

  The API is called when there is a requirement to read a register in the Motor Control IP of the Node System.
  The API creates burst packets which are sent to the Node System over EtherConnect. The header in the burst packet indicates that a particular packet is for Motor Status Read and for which nodes this packet intended. Once the burst packet is written in a FIFO module, it is sent to the Node System by a trigger of 1 to 0 signal in a Start Transaction Register. After the Node System has taken appropriate actions successfully, the Main System receives an interrupt, and it validates the value of the interrupt info register. Upon the confirmation of the value of the interrupt info register, this API returns a 0 value or a –1 if there is an error.

- PDM Data Fetch API

  int pdm_data_fetch_api(uint32_t total_size, uint32_t node_addr)

  The API is present in the main.c file. It needs two parameter:

  - total_size: the size of the PDM data required from user interface
  - node_addr: node select value sent in packet

  It returns the following integer values:

  - 0: if all tasks are successfully completed
  - –1: if there is an error

  The API is called when there is a requirement to read a bulk maintenance data from the Motor Control IP of the Node System.

  The maximum data that can be transferred in a single transaction from node to Main System is 256 bytes. Therefore, if the total_size is larger than 256 bytes, chunks of 256 bytes are requested one by one until the total_size requirement is met.

  This API first configures the DMA register by writing the destination base and destination end address in specific registers. The API creates burst packets which are sent to the Node System over EtherConnect. The header in the burst packet indicates that a particular packet is for PDM Data Fetch and for which node this packet intended. Once the burst packet is written in a FIFO module, it is sent to the Node System by a trigger of 1 to 0 signal in a *Start Transaction Register*. After the Node System completes the task successfully, the Main System receives an EtherConnect interrupt, and it validates the value of the interrupt info register. The value of the DMA status register is to be validated as confirmation of the same. A successful validation signifies that a single chunk of data is successfully written into the Main System memory. This process is repeated until all the chunks are received by the Main System.

  A final EtherConnect interrupt is then received from the Node System signifying the completion of the PDM data fetch command for the total_size. Upon confirmation of the value of the interrupt info register, this API returns with 0 value.

- PDM bulk Data Fetch API
  int pdm_bulk_data_fetch_api (uint32_t total_size, uint32_t node_addr)
  The API is present in the main.c file. It needs two parameters:
  - total_size: the size of the PDM data required from user interface
  - node_addr : node select value sent in packet

  It returns the following integer values:
  - 0: if all tasks are successfully completed
  - –1: if there is an error

  The API is called when there is a requirement to read a bulk maintenance data from the Motor Control IP of the Node System.
  This API is extended version of PDM Data Fetch API, as total size of data fetch depends on number of active nodes present in that chain.

4.1.2. OPCUA PubSub :

In PubSub model, a Publisher component, which can define DataSets that contain Variables or EventNotifiers.The Publisher publishes the DataSetMessages, which contain DataChanges or Events. The sender defines in Datasets what is sent, instead of the receiver. Publishers are the source of data, and the Subscribers consume that data. Communication in PubSub is message-based. Publishers send messages to a Message Oriented Middleware, Subscribers express interest in specific types of data, and process messages that contain this data. OPCUA PubSub supports two different Message Oriented Middleware variants, namely UDP based and Ethernet based protocol. Subscribers and Publishers use datagram protocols like UDP. The core component of the Message Oriented Middleware is a message broker. Subscribers and Publishers use standard messaging protocols like UDP or MQTT to communicate with the pub-sub.

- OPC UA defines two different Network types for PubSub.
  - Local Network – can use UDP Broadcast (or Unicast in some cases) or Ethernet APL. The messages are optimized binary UADP, which is defined in the OPC UA specifications. So, only OPC UA Subscribers can interpret the messages.
  - Message Queue Broker – can be an MQTT or AMQP broker, in practice. In this case, the messages are typically JSON messages, although UADP can be used for improved performance. The OPC Foundation has defined a standard content structure for the messages, but basically any JSON subscriber can interpret them.
    The Main System module implements following functions:
    - Generic variable Create_UADP_NetworkMessage ()
    - Generic variables UADP NetworkMessage_parse (**)**

4.1.3. Create_UADP_NetworkMessage

4.1.3.1. NetworkMessage Header:

The NetworkMessage is a container for DataSetMessages and includes information shared between DataSetMessages.

The following are the parameters of the Network Message Header:

- UADPVersion – The UADPVersion for this specification version is 1.
- UADPFlags – This flag enabled group header, Payload header, PublisherId.
- ExtendedFlags1 – The ExtendedFlags1 is omitted if bit 7 of the UADPFlags is false. The PublisherId type is of DataType Uint16.
- ExtendedFlags2 – The ExtendedFlags2 is omitted if bit 7 of the ExtendedFlags1 is false.
- PublisherId – The Id of the Publisher that sent the data. Valid DataType are Uintger and String.
- DataSetClassId – The DataSetClassId associated with the DataSets in the NetworkMessage.

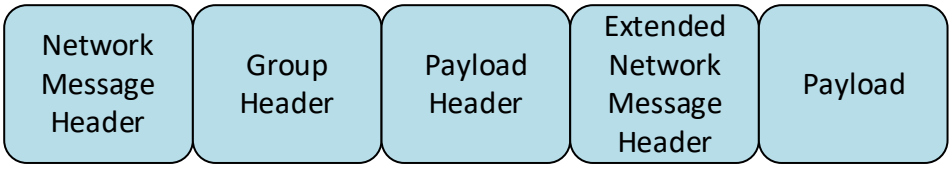

Figure 4.1. UADP Version

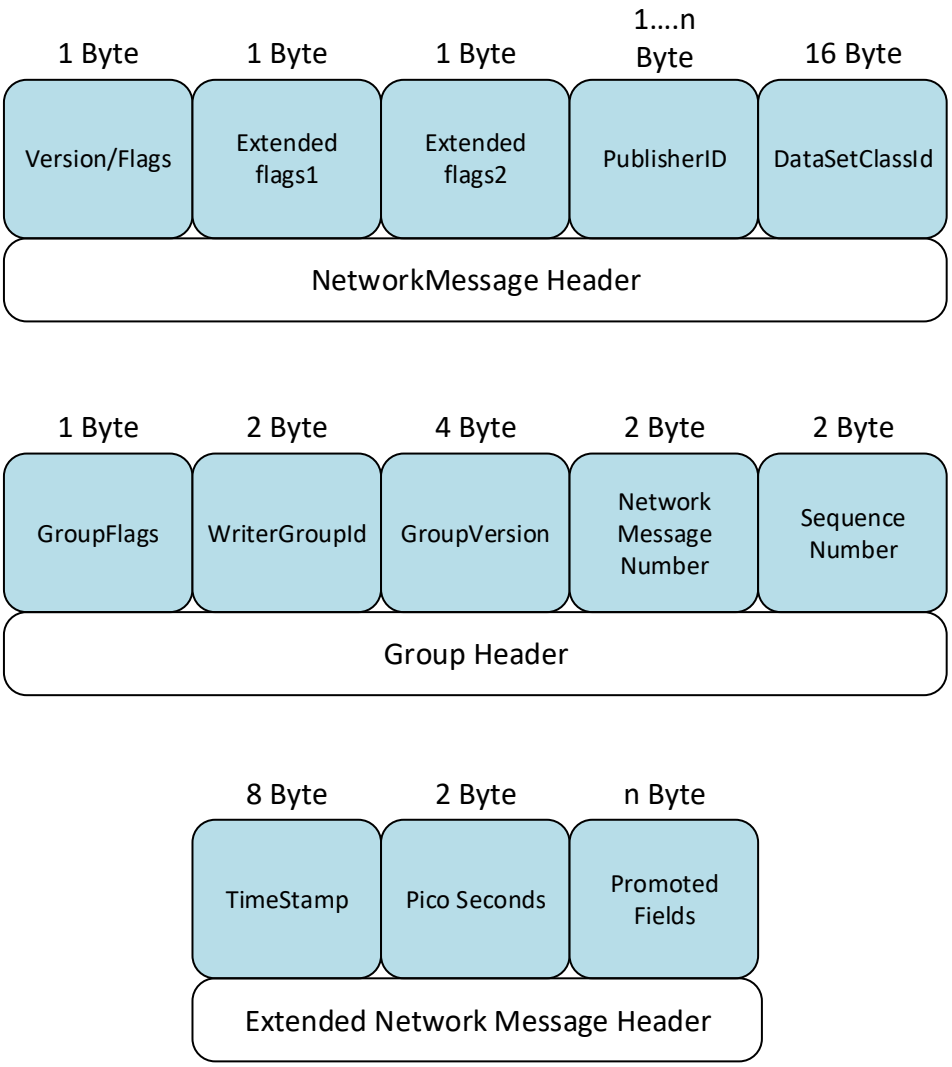

Figure 4.2. UADP Message Packet Header

4.1.4. GroupHeader

The group header is omitted if bit 5 of the UADPFlags is false.

- GroupFlags – GroupFlags is used for writerGroupId,GroupVersion enabled, NetworkMessageNumber enabled, SequenceNumber enabled.
- WriterGroupId – Unique id for the WriterGroup in the Publisher.
- GroupVersion – Version of the header and payload layout configuration of the NetworkMessages sent for the group.
- NetworkMessage Number – Unique number of a NetworkMessage combination of PublisherId and WriterGroupId within one PublishingInterval.
- SequenceNumber – Sequence number for the NetworkMessage.

4.1.5. Extended NetworkMessage Header

- Timestamp – The time the NetwrokMessage was created.
- PicoSeconds – Specifies the number of 10 picoseconds intervals which is added to the Timestamp.
- PromotedFields – PromotedFields are provided, the number of DataSetMessages in the Network Message is one.

4.1.5.1. Payload

Payload is defined with exact data of Node variables like nodeIds, requestType and these values. The UADP packet format size is 64 bytes, header size is 20 bytes, and Payload size is 44 bytes.

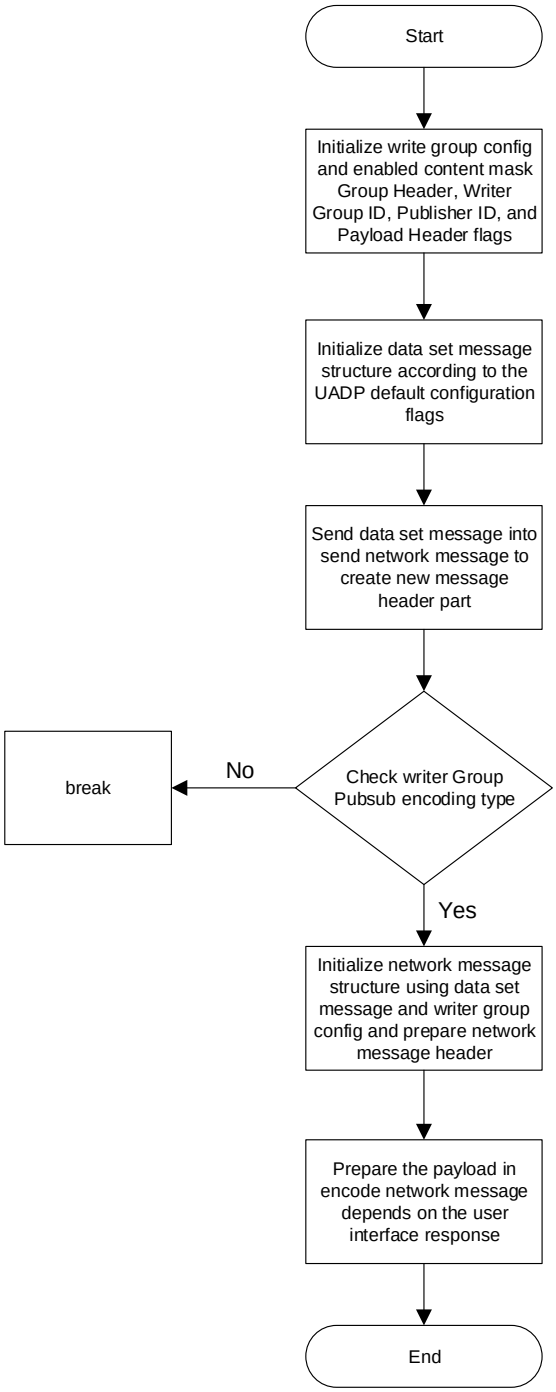

Figure 4.3. Create_UADP_NetworkMessage

UADP_NetworkMessage_parse :

This module parses the data received from the publisher. The publisher sends the 64 bytes OPCUA pubsub message, which holds the 20 bytes NetworkMessage header and, 44 Bytes payload. In payload data get the node ids and these node Ids are identify the method call or node variables or method variables, After identification create an UDP data reponse header,csv nodeid, request Type and value and writes the UDP data request on LPPDDR memory and get the UDP data response from lpddr memory. The parse data get method nodeIds then called the method according to the method nodeid like startmotor, stop motor, and power off.

```
void uadp_network_parse(unsigned int *Buffer);
```

The API is present in the UADP_NetworkMessage.c file. It gets the network message buffer from the user interface side.

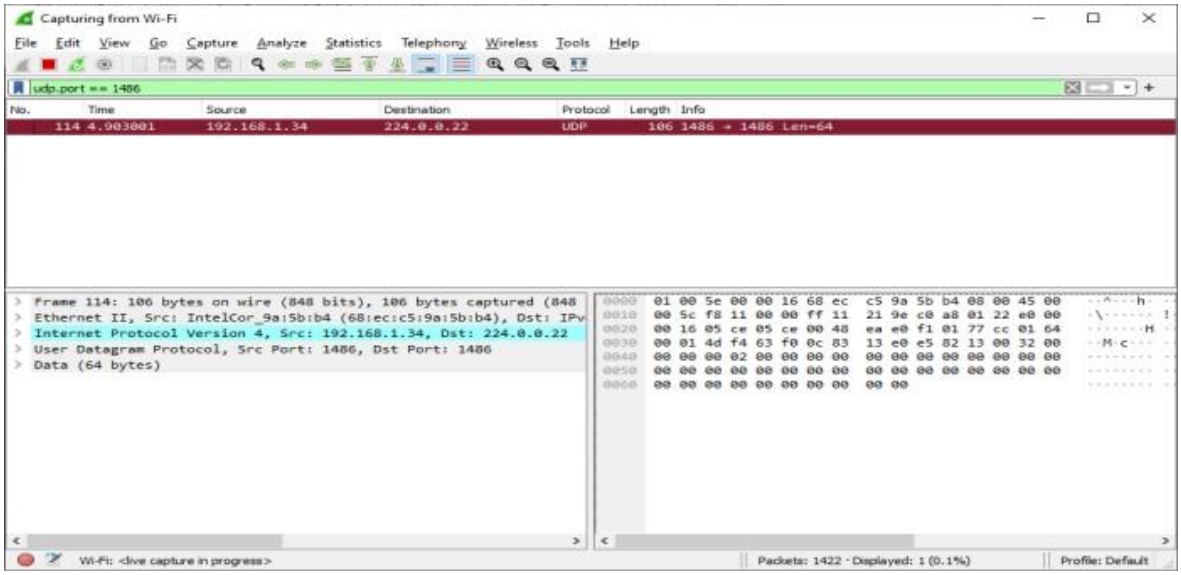

Figure 4.4. UADP Network Message Format

udp_response_func

This module writes the UDP data request to the LPDDR4 memory and gets the UDP data response from LPDDR4 memory.

```
void udp_response_func()
```

This API is present in the UADP_NetworkMessage.c file. It does not require any parameter.

method_callbacks

This module checks the method id and calls the method like start motor, stop motor, power off, update config, and run pdm.

```
void method_callbacks(unsigned char method)
```

This API is present in the rfl.c file. It gets the method nodeID parameter.

rfl_update_config

This module updates the motor variable configuration like rpm, breaker amps, number of Poles, max power.

```
void rfl_update_config()
```

This API is present in the rfl.c. file. It does not require any parameter.

start_motor

This function starts motor if motor is off or update target rpm of node.

```
void start_motor()
```

This API is present in the rfl.c file. It does not require any parameter.

stop_motor

This function stops the motor of all nodes. This function works when one of the motors is running.

```
void stop_motor()
```

This API is present in the rfl.c file. It does not require any parameter.

poweroff_motor

This function stops the power supply of all nodes. This function also works when one of the motors is running and is disabled if all motors are off.

```
void poweroff_motor()
```

This API is present in the rfl.c file. It does not require any parameter.

get_background

This function checks the Rpmlock, motor voltage, and motor status in background.

```
void get_background()
```

This API is present in the rfl.c file. It does not require any parameter.

run_pdm

This function collects the PDM data to generate the PDM image.

```
void run_pdm();
```

This API is present in the rfl.c file. It does not require any parameter.

4.2. Node System APIs

4.2.1. Tasks of the Node System

The Node System acts to control the Motor IP and get its status as commanded by the Main System. It communicates with the Main System by receiving commands through EtherConnect. It performs the actions and responds to the Main System with interrupts as acknowledgement for the tasks executed.

The tasks to be carried out by a master system can be categorized as follows:

- Communicate with the master system over EtherConnect
- As per the commands sent by the Main System, the Node System is supposed to either configure the motor, share the motor status, or share the bulk PDM data.
- Perform key functions

4.2.2. API Calls

- Main () function
  int main (void)
  - Upon a power on or a reset of the board, it is the job of the main function to initialize and configure the interrupts (EtherConnect, UART).
  - The main function then waits for the *ether_interrupt_flag* to get high. The EtherConnect ISR sets the flag, *ether_interrupt_flag* when a command is received from the Main System. When the main function finds that the flag is set, it reads the INTERRUPT STATUS register to decode which command is received. Based on the value of this register, the main function calls the appropriate functions.
- Node Perpherials init
  u08 general_init (void)
  - Upon a power on or a reset of the board, it is the job of the main function to initialize and configure the interrupts for UART, EtherConnect. It also initializes Modbus, SPI, and I^2C protocols.
- ISR1_EtherConnect
  *static void etherconnect_isr (void *ctx)*

    - The primary function of the EtherConnect ISR function is to set the interrupt flag, acknowledge/clear the interrupt and return an integer value. The EtherConnect interrupts are used as indicators of the receipt of a command sent by the Main System to the Node System. The IRQ value for EtherConnect is 0.
- Node Configuration API
  int node_config_api(void)
    - The API is present in the main.c file. It does not require any parameter.
    - It returns the following integer values:
        - 0: if all tasks are successfully completed
        - –1: if there is an error
    - The API is called when the main function receives a Node Config command in its Interrupt Status Register. This API reads the NODE ADDRESS register. This register contains an *address* of the peripheral (I^2C, Modbus, SPI, and Motor IP) which is supposed to be configured. Next, the NODE CONFIG DATA register is read. This register has the configuration *data*. This *data* is then written into the *address*. If there is a read or write error, the API returns a –1 value. Once completed, the API returns a 0 value.
- Node Status API
  int node_status_api(void)
    - The API is present in the main.c file. It does not require any parameter. This returns the following integer values:
        - 0: if all tasks are successfully completed
        - –1: if there is an error
    - The API is called whenever the main function receives a Node Status command in its Interrupt Status Register. This API reads the NODE ADDRESS register. This register contains an address of the Node peripherial (Modbus, SPI, I^2C, Motor IP) whose configuration value is supposed to be read. This address is then read and stored in a local variable data. This data is then written into the NODE STATUS register. If there is any read or write error, the API sends –1 value back. If everything goes okay, the API returns 0 value.
- PDM Data Fetch API
  int pdm_data_fetch_api(void)
    - The API is present in the main.c file. It does not require any parameter. This returns the following integer values:
        - 0: if all tasks are successfully completed
        - –1: if there is an error
    - The API first reads the *size* of PDM data required from the PDM ADDRESS register. It then writes the *base address* value and the *end address (base address + size)* value at the designated registers in the FIFO DMA Module. It then enables the FIFO DMA module by sending writing 0x00000003 first and then 0x00000000 to the FIFO DMA CONTROL register. Once done, it polls the DMA STATUS register for the indication of completion of the PDM data fetch. Once it receives the done value, it sets the DMA DONE INDICATE register. If there is any read or write error, the API sends –1 value back. If everything goes okay, the API returns 0 value.
- Node Peripheral APIs
    - I^2C Controller
      The following are the I^2C BSP functions used in the main.c file for writing and reading the I^2C target data:
        - uint8_t i2c_master_write(struct i2cm_instance × this_i2cm, uint16_t address,uint8_t data_size, uint8_t × data_buffer)
        - uint8_t i2c_master_read(struct i2cm_instance × this_i2cm, uint16_t address,uint8_t read_length, uint8_t × data_buffer)
    - SPI Controller
      The following are the SPI BSP functions used in the main.c file for writing and reading SPI target data:
        - uint8_t spi_master_write(struct spim_instance × this_spim,uint8_t data_size, uint8_t × data_buffer)
        - uint8_t spi_master_read(struct spim_instance × this_spim,uint8_t read_length, uint8_t × data_buffer)

    - Modbus RTU Master
      The following are the Modbus module functions used in the main.c file for writing and reading Modbus RTU slave data:
        - *eMBErrorCode eMBMasterInit(eMBMode eMode, void *dHUART, ULONG ulBaudRate, void *dHTIM)*
          This function initializes the ASCII or RTU module and calls the init functions of the porting layer to prepare the hardware. Note that the receiver is still disabled and no Modbus frames are processed until eMBMasterEnable() is called.
        - *eMBErrorCode eMBMasterPoll(void)*
          This function must be called periodically. The timer interval required is given by the application dependent Modbus slave timeout. Internally thefunction calls xMBMasterPortEventGet() and waits for an event from the receiver or transmitter state machines.
- unsigned int modbus_req (unsigned int mod_addr, unsigned int mod_data)
    - This function parses the data received from Main system and fetch slave id command type and data from it. This calls the functions below based on the command type.
        - *eMBMasterReqWriteHoldingRegister (slaveid, regnum, regdata, timeout)*
        - *eMBMasterReqWriteCoil (slaveid, regnum, regdata, timeout)*
- OPCUA INIT
    - void opcua_init(void)
        - This API is called to initialize the opuca header format. In this API, store the publisher ID and writer ID these IDs are used into the pub-sub communication.
- OPCUA packet parse
    - void opcua_etherconnect_parse(void)
        - This API parse the OPCUA packet which gets from the ethernet to have the information about nodes. Nodes information like node_Id, request_type and payload.
- OPCUA header response
    - void opcua_header_response_loaded(unsigned int *response_packet)
        - This API is loaded the default UADP network message header, which have the information about the writer ID, publisher ID, and writer group ID and use of these IDs in the OPCUA pub-sub communication.

5. Communications

This section describes the communications between the host to the Main System and the communication between the Main System and the Node Systems. Detailed breakdown of message vocabulary and packet structure may be covered in a separate document.

5.1. Communication between Host and Main System

Initially, this connection is implemented using the Ethernet interface. Most of the messages must be ASCII to facilitate debugging using the terminal program on the Host.

5.1.1. Messages from Host to Main System

- Motor Configuration and Control
- PDM Configuration and Control
- Request Motor Status
- Request PDM Status
- Request PDM Data – Normal
- Request PDM Data – Extended

5.1.2. Messages from Main System to Host

- System Information (Link Status, Connected Nodes, Local Delay of Nodes, and others)
- Motor Status
- PDM Status
- PDM Data – Normal
- PDM Data – Extended

5.2. Communication between Main System and Node System(s)

The physical connection between the Main System and Node System is implemented using Ethernet Cat-5 cables. The physical connection between the first Node System and subsequent Node System(s) also uses Ethernet Cat-5 cables, in a daisy-chain fashion for both chains.

5.2.1. Messages from Main System to Node System

- Motor Configuration and Control
- PDM Configuration and Control
- Request Motor Status
- Request PDM Status
- Request PDM Data – Normal
- Request PDM Data – Extended

5.2.2. Messages from Node System to Main System

- Node Information (Link Status, Connected Nodes, and Local Delay)
- Motor Status
- PDM Status
- PDM Data – Normal
- PDM Data – Extended

Appendix A. Predictive Maintenance with TensorFlow Lite

A.1. Setting Up the Linux Environment for Neural Network Training

This section describes the steps for setting up NVIDIA GPU drivers and/or libraries for 64-bit Ubuntu 16.04 OS. The NVIDIA library and TensorFlow version is dependent on the PC and Ubuntu/Windows version.

A.1.1. Installing the NVIDIA CUDA and cuDNN Library for Machine Learning Training on GPU

A.1.1.1. Installing the CUDA Toolkit

To install the CUDA toolkit, run the following commands in the order specified below:

```
$ curl -O
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/cuda-
repo-ubuntu1604_10.1.105-1_amd64.deb
```

```
$ curl -O https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  2832  100  2832    0     0   2204      0  0:00:01  0:00:01 --:--:--  2205
```

Figure A.1. Download CUDA Repo

```
$ sudo dpkg –I ./cuda-repo-ubuntu1604_10.1.105-1_amd64.debA
```

```
$ sudo dpkg -i ./cuda-repo-ubuntu1604_10.1.105-1_amd64.deb
Selecting previously unselected package cuda-repo-ubuntu1604.
(Reading database ... 5287 files and directories currently installed.)
Preparing to unpack .../cuda-repo-ubuntu1604_10.1.105-1_amd64.deb ...
Unpacking cuda-repo-ubuntu1604 (10.1.105-1) ...
Setting up cuda-repo-ubuntu1604 (10.1.105-1) ...

The public CUDA GPG key does not appear to be installed.
To install the key, run this command:
sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
```

Figure A.2. Install CUDA Repo

```
$ sudo apt-key adv --fetch-keys
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.p
ub
```

```
$ sudo apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
Executing: /tmp/tmp.a2QZZnTMUX/gpg.1.sh --fetch-keys
http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64/7fa2af80.pub
gpg: key 7FA2AF80: public key "cudatools <cudatools@nvidia.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

Figure A.3. Fetch Keys

```
$sudo apt-get update
```

```
$ sudo apt-get update
Ign:1 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  InRelease
Hit:2 http://archive.ubuntu.com/ubuntu xenial InRelease
Get:3 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Release [697 B]
Get:4 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Release.gpg [836 B]
Hit:5 http://archive.ubuntu.com/ubuntu xenial-updates InRelease
Hit:6 http://security.ubuntu.com/ubuntu xenial-security InRelease
Hit:7 http://archive.ubuntu.com/ubuntu xenial-backports InRelease
Ign:8 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Packages
Get:8 http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64  Packages [428 kB]
Fetched 429 kB in 1s (386 kB/s)
Reading package lists... Done
```

Figure A.4. Update Ubuntu Packages Repositories

```
$ sudo apt-get install cuda-9-0
```

```
$ sudo apt-get install cuda-9-0
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Figure A.5. CUDA Installation

A.1.1.2. Installing the cuDNN

To install the cuDNN:

1. Create NVIDIA developer account: https://developer.nvidia.com.
2. Download cuDNN lib: https://developer.nvidia.com/compute/machine-learning/cudnn/secure/v7.1.4/prod/9.0_20180516/cudnn-9.0-linux-x64-v7.1
3. Execute the commands below to install cuDNN

```
$ tar xvfcudnn-9.0-linux-x64-v7.1.tgz
$ sudo cp cuda/include/cudnn.h/usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudochmoda+r/usr/local/cuda/include/cudnn.h/usr/local/cuda/lib64/libcudnn*
```

```
$ tar xcf cudnn-9.0-linux-x64-v7.1.tgz
cuda/include/cudnn.h
cuda/NVIDIA_SLA_cuDNN_Support.txt
cuda/lib64/libcudnn.so
cuda/lib64/libcudnn.so.7
cuda/lib64/libcudnn.so.7.1.4
cuda/lib64/libcudnn_static.a

$ sudo cp cuda/include/cudnn.h /usr/local/cuda/include
$ sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
$ sudo chmod a+r /usr/local/cuda/include/cudnn.h /usr/local/cuda/lib64/libcudnn*
```

Figure A.6. cuDNN Library Installation

A.1.2. Setting Up the Environment for Training and Model Freezing Scripts

This section describes the environment setup information for training and model freezing scripts for 64-bit Ubuntu 16.04. Anaconda provides one of the easiest ways to perform machine learning development and training on Linux.

A.1.2.1. Installing the Anaconda Python

To install the Anaconda and Python 3:

1. Go to the https://www.anaconda.com/products/individual#download website.
2. Download Python3 version of Anaconda for Linux.
3. Run the command below to install the Anaconda environment:
   ```
   $  sh  Anaconda3-2019.03-Linux-x86_64.sh
   ```
   Note: Anaconda3-**<version>**-Linux-x86_64.sh, version may vary based on the release.

```
(base) $ sh Anaconda3-2020.07-Linux-x86_64.sh

Welcome to Anaconda3 2020.07

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>
```

Figure A.7. Anaconda Installation

4. Accept the license.

```
Do you accept the license terms? [yes|no]
[no] >>> yes
```

Figure A.8. Accept License Terms

5. Confirm the installation path. Follow the instruction onscreen if you want to change the default path.

```
[no] >>> yes


Anaconda3 will now be installed into this location:
/home/user/anaconda3

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below

[/home/user/anaconda3] >>> /home/user/anaconda3
```

Figure A.9. Confirm/Edit Installation Location

6. After installation, enter *no* as shown in Figure A.10.

```
Preparing transaction: done
Executing transaction: done
installation finished.
Do you wish the installer to initialize Anaconda3
by running conda init? [yes|no]
[no] >>>  no
```

Figure A.10. Launch/Initialize Anaconda Environment on Installation Completion

A.1.3. Installing the TensorFlow Version 1.15

To install the TensorFlow version 1.15:

1. Activate the Anaconda environment by running the command below:
```
$ source <conda directory>/bin/activate
```

```
$ source anaconda3/bin/activate
(base) ~$
```

Figure A.11. Anaconda Environment Activation

2. Install the TensorFlow by running the command below:
```
$ conda install tensorflow-gpu==1.15.0
```

```
$ conda install tensorflow-gpu==1.15.0
Collecting package metadata (current_repodata.json): done
Solving environment: failed with initial frozen solve. Retrying with flexible solve.
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3/

  added / updated specs:
    - tensorflow-gpu==1.15.0
```

Figure A.12. TensorFlow Installation

3. After installation, enter **Y** as shown in Figure A.13.

```
  tensorboard         pkgs/main/noarch::tensorboard-1.15.0-pyhb230dea_0
  tensorflow          pkgs/main/linux-64::tensorflow-1.15.0-mkl_py36h4920b83_0
  tensorflow-base     pkgs/main/linux-64::tensorflow-base-1.15.0-mkl_py36he1670d9_0
  tensorflow-estima~  pkgs/main/noarch::tensorflow-estimator-1.15.1-pyh2649769_0
  termcolor           pkgs/main/linux-64::termcolor-1.1.0-py36h06a4308_1
  webencodings        pkgs/main/linux-64::webencodings-0.5.1-py36_1
  werkzeug            pkgs/main/noarch::werkzeug-0.16.1-py_0
  wrapt               pkgs/main/linux-64::wrapt-1.12.1-py36h7b6447c_1
  zipp                pkgs/main/noarch::zipp-3.4.0-pyhd3eb1b0_0


Proceed ([y]/n)? y
```

Figure A.13. TensorFlow Installation Confirmation

Figure A.14 shows TensorFlow installation is complete.

```
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

Figure A.14. TensorFlow Installation Completion

A.1.4. Installing the Python Package

To install the Python package:

1. Install Easydict by running the command below:
```
$ conda install -c conda-forge easydict
```

```
(base) $ conda install -c conda-forge easydict
Solving environment: done
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - easydict
```

Figure A.15. Easydict Installation

2. Install Joblib by running the command below:
```
$ conda install joblib
```

```
(base) $ conda install joblib
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - joblib
```

Figure A.16. Joblib Installation

3. Install Keras by running the command below:
```
$ conda install keras
```

```
(base) $ conda install keras
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - keras
```

Figure A.17. Keras Installation

4. Install OpenCV by running the command below:
```
$ conda install opencv
```

```
(base) $ conda install opencv
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - opencv
```

Figure A.18. OpenCV Installation

5. Install Pillow by running the command below:
```
$ conda install pillow
```

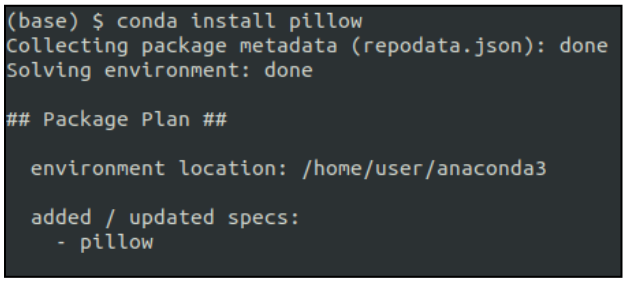

Figure A.19. Pillow Installation

A.2. Creating the TensorFlow Lite Conversion Environment

To create a new Anaconda environment and install tensorflow=2.2.0:

1. Create a new Anaconda environment.
```
$ conda create -n <New Environment Name> python=3.6
```
2. Activate new created environment.
```
$ conda activate <New Environment Name>
```
3. Install Tensorflow 2.2.0.
Note: We have noticed output difference in Tensorflow(2.2.0) and Tensorflow-gpu(2.2.0) in terms of tflite size. It is recommended to use TensorFlow (2.2.0).
```
$ conda install tensorflow=2.2.0
```
4. Install opencv.
```
$conda install opencv
```

A.3. Preparing the Dataset

This section describes the steps and guidelines used to prepare the dataset for training the predictive maintenance.
Note: In the following sections, Lattice provides guidelines and/or examples that can be used as references for preparing the dataset for the given use cases. Lattice is not recommending and/or endorsing any dataset(s). It is recommended that customers gather and prepare their own datasets for their specific end applications.

A.3.1. Dataset Information

In the predictive maintenance demonstration, there are three classes: bad, Normal, and unknown. The dataset must be organized as shown in Figure A.20. The *0* folder contains bad motor data and the *1* folder contains normal motor data.

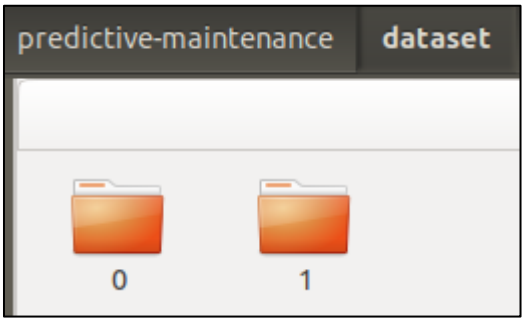

Figure A.20. Predictive Maintenance Dataset

A.4. Preparing the Training Code

Notes:

- Training and freezing code uses TensorFlow 1.15.0 since some of the APIs used in training code are not available in Tensorflow 2.x.
- For the TensorFlow Lite conversion in the TensorFlow Lite Conversion and Evaluation section**,** TensorFlow 2.2.0 is used.

A.4.1. Training Code Structure

Download the Lattice predictive maintenance demo training code. Its directory structure is shown in Figure A.21.

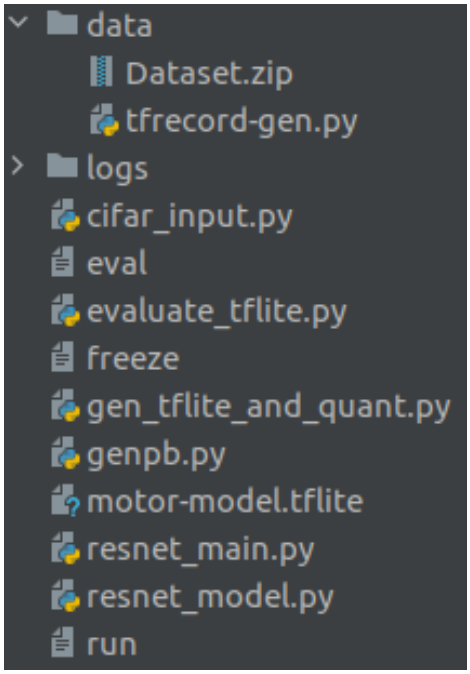

Figure A.21. Training Code Directory Structure

A.4.2. Generating tfrecords from Augmented Dataset

This demo only takes tfrecords of a specific format for input. As such, generate the tfrecords first. Run the command below to generate tfrecords from input dataset.

```
$ python tfrecord-gen.py -i <Input_augmented_dataset_root> -o <Output_tfrecord_path>
```

The input directory should follow the structure shown in Figure A.21.

A.4.3 Neural Network Architecture

This section provides information on the Convolution Neural Network Configuration of the Predictive Maintenance design.

Table A.1. Predictive Maintenance Training Network Topology

Input Gray Scale Image (64×64×1)		
Fire1	Conv3x3 – 8	Conv3×3 - # where: • Conv3×3 – 3 × 3 Convolution filter Kernel size • # - The number of filters For example, Conv3×3 - 8 = 8 3 × 3 convolution filter Batchnorm: Batch Normalization FC - # where: • FC – Fully connected layer • # - The number of outputs
	Batchnorm	
	ReLU	
	Maxpool	
Fire2	Conv3×3 – 8	
	Batchnorm	
	ReLU	
Fire3	Conv3×3 – 16	
	Batchnorm	
	ReLU	
	Maxpool	
Fire4	Conv3×3 – 16	
	Batchnorm	
	ReLU	
Fire5	Conv3×3 – 16	
	Batchnorm	
	ReLU	
	Maxpool	
Fire6	Conv3×3 – 22	
	Batchnorm	
	ReLU	
Fire7	Conv3×3 – 24	
	Batchnorm	
	ReLU	
	Maxpool	
Dropout	Dropout - 0.80	
logit	FC – (3)	

In Table A.1, Layer contains Convolution (conv), batch normalization (BN), ReLU, pooling, and dropout layers. Output of layer logit is (Broken [0], Normal [1], Unknown [2]) 3 values.

- Layer information
  - Convolutional Layer

    In general, the first layer in a CNN is always a convolutional layer. Each layer consists of number of filters (sometimes referred as kernels) which convolves with input layer/image and generates activation map (such as feature map). This filter is an array of numbers (the numbers are called weights or parameters). Each of these filters can be thought of as feature identifiers, like straight edges, simple colors, and curves and other high-level features. For example, the filters on the first layer convolve around the input image and “activate” (or compute high values) when the specific feature (say curve) it is looking for is in the input volume.
  - ReLU (Activation Layer)
    After each conv layer, it is convention to apply a nonlinear layer (or activation layer) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the conv layers (just element wise multiplications and summations).In the past, nonlinear functions like tanh and sigmoid were used, but researchers found out that ReLU layers work far better because the network is able to train a lot faster (because of the computational efficiency) without making a significant difference to the accuracy. The ReLU layer applies the function f(x) = max (0, x) to all values in the input volume. In basic terms, this layer just changes all the negative activations to 0. This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the conv layer.
  - Pooling Layer

    After some ReLU layers, programmers may choose to apply a pooling layer. It is also referred to as a down sampling layer. In this category, there are also several layer options, with Maxpooling being the most popular. This basically takes a filter (normally of size 2×2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every sub region that the filter convolves around.

    The intuitive reasoning behind this layer is that once the user knows that a specific feature is in the original input volume (a high activation value results), its exact location is not as important as its relative location to the other features. As you can imagine, this layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes. The first is that the number of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it controls over fitting. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets. A symptom of over fitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.
  - Batchnorm Layer

    Batch normalization layer reduces the internal covariance shift. To train a neural network, perform pre-processing to the input data. For example, the user can normalize all data so that it resembles a normal distribution (that means, zero mean and a unitary variance). Reason being preventing the early saturation of non-linear activation functions like the sigmoid function, assuring that all input data is in the same range of values.
    But the problem appears in the intermediate layers because the distribution of the activations is constantly changing during training. This slows down the training process because each layer must learn to adapt themselves to a new distribution in every training step. This problem is known as internal covariate shift.
    Batch normalization layer forces the input of every layer to have approximately the same distribution in every training step by following below process during training time:
    - Calculate the mean and variance of the layers input.
    - Normalize the layer inputs using the previously calculated batch statistics.
    - Scales and shifts to obtain the output of the layer.

    This makes the learning of layers in the network more independent of each other and allows you to be carefree about weight initialization, works as regularization in place of dropout and other regularization techniques.

- Drop-out Layer
  Dropout layers have a very specific function in neural networks. After training, the weights of the network are so tuned to the training examples they are given that the network does not perform well when given new examples. The idea of dropout is simplistic in nature. This layer *drops out* a random set of activations in that layer by setting them to zero. It forces the network to be redundant. That means the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network is not getting too "fitted" to the training data and thus helps alleviate the over fitting problem. An important note is that this layer is only used during training, and not during test time.
- Fully-connected Layer
  This layer basically takes an input volume (whatever the output is of the conv or ReLU or pool layer preceding it) and outputs an N dimensional vector where N is the number of classes that the program must choose from.
- Quantization
  Quantization is a method to bring the neural network to a reasonable size, while also achieving high performance accuracy. This is especially important for on-device applications, where the memory size and number of computations are necessarily limited. Quantization for deep learning is the process of approximating a neural network that uses floating-point numbers by a neural network of low bit width numbers. This dramatically reduces both the memory requirement and computational cost of using neural networks.

The above architecture provides nonlinearities and preservation of dimension that help to improve the robustness of the network and control over fitting.

A.4.4. Training Code Overview

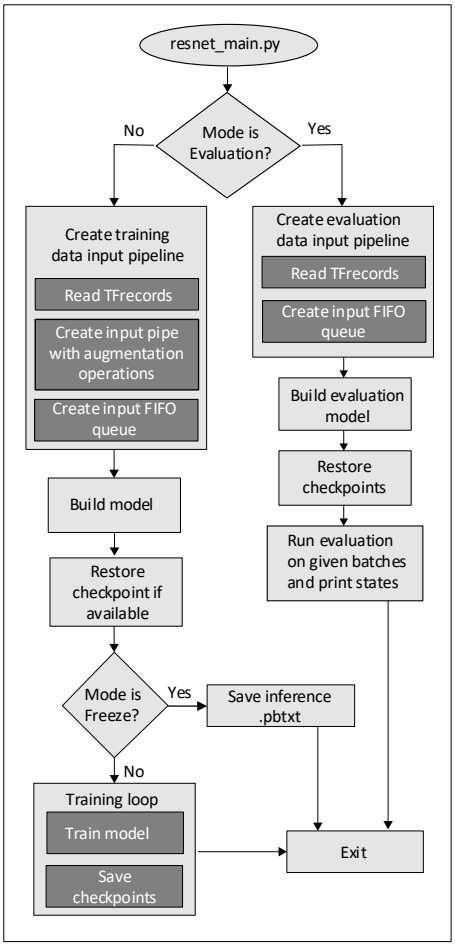

Figure A.22. Training Code Flow Diagram

A.4.4.1. Configuring Hyper-Parameters

```
if FLAGS.mode == 'train':
    batch_size = 128
elif FLAGS.mode == 'eval':
    batch_size = 100
if FLAGS.mode == 'freeze':
    batch_size = 1

if FLAGS.dataset == 'signlang':
    num_classes = 2 + 1  # broken, normal, unknown

hps = resnet_model.HParams(batch_size=batch_size,
                           num_classes=num_classes,
                           min_lrn_rate=0.0001,
                           lrn_rate=0.1,
                           num_residual_units=5,  # 2*3*this
                           use_bottleneck=False,
                           weight_decay_rate=0.0002,
                           relu_leakiness=0.1,
                           optimizer='mom'  # sgd, mom, adam,
                           )  # resNet enable
```

Figure A.23. Code Snippet: Hyper Parameters

- Set number of classes in *num_classes* (default = 3).
- Change batch size for specific mode if required.
- hps: it contains list of hyper parameters for custom resnet backbone and optimizer.

A.4.4.2. Creating Training Data Input Pipeline

```
images, labels = cifar_input.build_input(
    FLAGS.dataset, FLAGS.train_data_path, hps.batch_size, FLAGS.mode, FLAGS.gray, hps[1])
```

Figure A.24. Code Snippet: Build Input

- *build_input* () from *cifer_input.py* reads Tfrecords and creates some augmentation operations before pushing the input data to FIFO queue.
  - *FLAGS.dataset :* dataset type (signlang)
  - FLAGS.train_data_path: input path to tfrecords
  - FLAGS.batch_size: training batch size
  - FLAGS.mode: train or eval
  - *FLAGS.gray:* True if model is of 1 channel otherwise False
  - *hps[1]:* num_classes configured in model hyper parameters

Read tfrecords

```
if dataset == 'signlang':  # TFRecord format
    reader = tf.TFRecordReader()
    _, serialized_example = reader.read(file_queue)
    features = tf.parse_single_example(
        serialized_example,
        features={
            'image/height': tf.FixedLenFeature([], tf.int64),
            'image/width': tf.FixedLenFeature([], tf.int64),
            'image/class/label': tf.FixedLenFeature([], tf.int64),
            'image/encoded': tf.FixedLenFeature([], tf.string)
        }
    )
```

Figure A.25. Code Snippet: Parse tfrecords

- Above snippet reads tfrecord files and parse its features that are height, width, label, and image.

Converting Image to Grayscale and Scaling the Image

```
if gray:  # Gray color
    image = tf.image.rgb_to_grayscale(image)
    depth = 1
```

Figure A.26. Code Snippet: Convert Image to Grayscale

- Convert RGB image to gray scale if gray flag is true.

```
channels = tf.unstack(image, axis=-1)

if gray:
    image = tf.stack([channels[0]], axis=-1)
else:
    # RGB to BGR Conversion
    image = tf.stack([channels[2], channels[1], channels[0]], axis=-1)

# image /= 128.0  # [0, 2)
image = image - 128.0
```

Figure A.27. Code Snippet: Convert Image to BGR and Scale the Image

- Unstack channel layers and convert to BGR format if the image mode is not gray. The RGB is converted to BGR because the iCE40 works on BGR image.
- Divide every element on image with 128 so that the values can be scaled to 0-2 range.

Creating Input Queue

```
example_queue = tf.RandomShuffleQueue(
    capacity=16 * batch_size,
    min_after_dequeue=8 * batch_size,
    dtypes=[tf.float32, tf.int32],
    shapes=[[image_size, image_size, depth], [1]])
num_threads = 16
```

Figure A.28. Code Snippet: Create Queue

- tf.RandomShuffleQueue is queue implementation that dequeues elements in random order.

```
example_enqueue_op = example_queue.enqueue([image, label])
tf.train.add_queue_runner(tf.train.queue_runner.QueueRunner(
    example_queue, [example_enqueue_op] * num_threads))

# Read 'batch' labels + images from the example queue.
images, labels = example_queue.dequeue_many(batch_size)
labels = tf.reshape(labels, [batch_size, 1])
indices = tf.reshape(tf.range(0, batch_size, 1), [batch_size, 1])
labels = tf.sparse_to_dense(
    tf.concat(values=[indices, labels], axis=1),
    [batch_size, num_classes], 1.0, 0.0)
```

Figure A.29. Code Snippet: Add Queue Runners

- Above snippet enqueues images and labels to the *RandomShuffleQueue* and add queue runners. This directly feeds data to network.

A.4.4.3. Model Building

CNN Architecture

```
model = resnet_model.ResNet(hps, images, labels, FLAGS.mode)
model.build_graph()
```

Figure A.30. Code Snippet: Create Model

- *Build_graph ()* method creates training graph or training model using given configuration.
- *Build_graph* creates model with seven fire layers followed by dropout layer and fully connected layers. Where each fire layer contains convolution, relu as activation, batch normalization, and max pooling (in Fire 1, 3, 5 & 7 only). Fully connected layer provides the final output.

```
fire1 = self._vgg_layer('fire1', self._images, oc=depth[0], freeze=False, pool_en=True,
                        bias_on=bias_on, phase_train=phase_train)
```

Figure A.31. Code Snippet: Fire Layer

- The following are the arguments of the *_vgg_layer*:
  - First argument is name of the block.
  - Second argument is input node to new fire block.
  - *oc*: output channels are the number of filters of the convolution.
  - *freeze*: setting weighs are trainable or not.
  - *w_bin*: Quantization parameter for convolution
  - *a_bin*: quantization parameter for activation binarization(relu).
  - *pool_en:* flag to include Maxpool in firelayer.
  - *min_rng, max_rng*: Setting maximum and minimum values of quantized activation. Default values for *min_rng = 0.0* and *max_rng = 2.0*.
  - *bias_on*: Sets bias add operation in graph if true.
  - *phase_train*: Argument to generate graph for inference and training.

```
def _vgg_layer(self, layer_name, inputs, oc, stddev=0.01, freeze=False, w_bin=16, a_bin=16, pool_en=True,
               min_rng=-0.5, max_rng=0.5, bias_on=True, phase_train=True):
    with tf.variable_scope(layer_name):
        net = self._conv_layer('conv3x3', inputs, filters=oc, size=3, stride=1, xavier=False,
                               padding='SAME', stddev=stddev, freeze=freeze, relu=False, w_bin=w_bin,
                               bias_on=bias_on)
        tf.summary.histogram('before_bn', net)
        net = self._batch_norm_tensor2('bn', net, phase_train=phase_train)  # BatchNorm
        tf.summary.histogram('before_relu', net)
        net = self.binary_wrapper(net, a_bin=a_bin, min_rng=min_rng, max_rng=max_rng)  # ReLU
        tf.summary.histogram('after_relu', net)
        if pool_en:
            pool = self._pooling_layer('pool', net, size=2, stride=2, padding='SAME')
        else:
            pool = net
        tf.summary.histogram('pool', pool)

        return pool
```

Figure A.32. Code Snippet: Convolution Block

- In the *resnet_model.py* file, the basic network construction blocks are implemented in specific functions as below:
  - Convolution – _conv_layer
  - Batch normalization – _batch_norm_tensor2
  - ReLU – binary_wrapper
  - Maxpool – _pooling_layer
- _conv_layer
  - Contains code to create convolution block. Which contains kernel variable, variable initializer, quantization code, convolution operation and ReLU if argument *relu* is True.
- _batch_norm_tensor2
  - Contains code to create batch-normalization operation for both training and inference phase.
- Binary_wrapper
  - Used for quantized activation with ReLU.
- _pooling_layer
  - Adds Max pooling with given kernel-size and stride size to training and inference graph.

Feature Depth of Fire Layer

```
depth = [8, 8, 16, 16, 16, 22, 24]
```

Figure A.33. Code Snippet: Feature Depth Array for Fire Layers

- List *depth* contains feature depth for seven fire layers in network.

```
fire1 = self._vgg_layer('fire1', self._images, oc=depth[0], freeze=False, pool_en=True,
                        bias_on=bias_on, phase_train=phase_train)
fire2 = self._vgg_layer('fire2', fire1, oc=depth[1], freeze=False, pool_en=False,
                        bias_on=bias_on, phase_train=phase_train)
fire3 = self._vgg_layer('fire3', fire2, oc=depth[2], freeze=False, pool_en=True,
                        bias_on=bias_on, phase_train=phase_train)
fire4 = self._vgg_layer('fire4', fire3, oc=depth[3], freeze=False, pool_en=False,
                        bias_on=bias_on, phase_train=phase_train)
fire5 = self._vgg_layer('fire5', fire4, oc=depth[4], freeze=False, pool_en=True,
                        bias_on=bias_on, phase_train=phase_train)
fire6 = self._vgg_layer('fire6', fire5, oc=depth[5], freeze=False, pool_en=False,
                        bias_on=bias_on, phase_train=phase_train)
fire7 = self._vgg_layer('fire7', fire6, oc=depth[6], freeze=False, pool_en=True,
                        bias_on=bias_on, phase_train=phase_train)
if phase_train:
    fire_o = tf.nn.dropout(fire7, 0.8)
else:
    fire_o = tf.nn.dropout(fire7, 1)
logits = self._fc_layer('logit', fire_o, self.hps.num_classes, flatten=True, relu=False, xavier=True)
```

Figure A.34. Code Snippet: Forward Graph Fire Layers

Loss Function and Optimizers

```
with tf.variable_scope('costs'):
    xent = tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=self.labels)
    self.cost = tf.reduce_mean(xent, name='xent')
    self.cost += self._decay()

    tf.summary.scalar('cost', self.cost)
```

Figure A.35. Code Snippet: Loss Function

- Model uses *softmax_cross_entropy_with_logitds* because the labels are in form of class index.

```
if self.hps.optimizer == 'sgd':
    optimizer = tf.train.GradientDescentOptimizer(self.lrn_rate)
elif self.hps.optimizer == 'mom':
    optimizer = tf.train.MomentumOptimizer(self.lrn_rate, 0.9)
elif self.hps.optimizer == 'adam':
    optimizer = tf.train.AdamOptimizer(self.lrn_rate)
elif self.hps.optimizer == 'rmsprop':
    optimizer = tf.train.RMSPropOptimizer(self.lrn_rate, decay=0.9, momentum=0.9, epsilon=1.0)
```

Figure A.36. Code Snippet: Optimizers

- There are four options for selecting optimizers. In this model, use the *mom* optimizer as default.

A.4.4.4. Restore Checkpoints

Checkpoints are restored from log directory and then starts training from that checkpoint if checkpoints exist in log directory.

```
try:
    ckpt_state = tf.train.get_checkpoint_state(FLAGS.ref_log_root)
    if not (ckpt_state and ckpt_state.model_checkpoint_path):
        tf.logging.info('No model to eval yet at %s', FLAGS.ref_log_root)
    else:
        tf.logging.info('Loading checkpoint %s', ckpt_state.model_checkpoint_path)
        saver.restore(sess, ckpt_state.model_checkpoint_path)
except Exception as e:
    tf.logging.error('Cannot restore checkpoint: %s', e)
```

Figure A.37. Code Snippet: Restore Checkpoints

A.4.4.5. Saving .pbtxt

If mode is *freeze* it saves the inference graph (model) as *.pbtxt* file. The *.pbtxt* file is used later for freezing.

```
if FLAGS.mode == "freeze":
    tf.train.write_graph(sess.graph_def, FLAGS.log_root, "model.pbtxt")
    print("Saved model.pbtxt at", FLAGS.log_root)
    sys.exit()
tf.train.start_queue_runners(sess)
```

Figure A.38. Code Snippet: Save .pbtxt

A.4.4.6. Training Loop

```
with tf.train.MonitoredTrainingSession(
        checkpoint_dir=FLAGS.log_root,
        hooks=[logging_hook, _LearningRateSetterHook()],
        chief_only_hooks=[summary_hook],
        save_summaries_steps=0,
        save_checkpoint_steps=FLAGS.ckptinterval,
        config=tf.ConfigProto(allow_soft_placement=True)) as mon_sess:
    confusion_matrix = np.zeros((hps[1], hps[1]))
    while not mon_sess.should_stop() and mon_sess.run(model.global_step) < FLAGS.maxsteps:
        _, confusion = mon_sess.run([model.train_op, model.confusion_matrix])
        confusion_matrix = np.add(confusion_matrix, np.array(confusion))
        if mon_sess.run(model.global_step) % FLAGS.ckptinterval == 0 and mon_sess.run(model.global_step) != 0:
            print("Confusion_Matrix :\n {}".format(confusion_matrix.astype(np.int)))
            confusion_matrix = np.zeros((hps[1], hps[1]))
```

Figure A.39. Code Snippet: Training Loop

- *MonitoredTrainingSession* utility sets proper session initializer/restorer. It also creates hooks related to checkpoint and summary saving. For workers, this utility sets proper session creator which waits for the chief to initialize/restore. Refer to tf.compat.v1.train.MonitoredSession for more information.
- _LearningRateSetterHook

```
def after_run(self, run_context, run_values):
    train_step = run_values.results
    if train_step < 20000:
        self._lrn_rate = 0.1
    elif train_step < 35000:
        self._lrn_rate = 0.01
    elif train_step < 50000:
        self._lrn_rate = 0.001
    elif train_step < 60000:
        self._lrn_rate = 0.0001
    else:
        self._lrn_rate = 0.00001
```

Figure A.40. Code Snippet: _ LearningRateSetterHook

- This hook sets learning rate based on training steps performed.

- Summary_hook

```
summary_hook = tf.train.SummarySaverHook(
    save_steps=100,
    output_dir=FLAGS.train_dir,
    summary_op=tf.summary.merge([model.summaries,
                                 tf.summary.scalar('Precision', precision)]))
```

Figure A.41. Code Snippet: Save Summary for Tensorboard

- Saves tensorboard summary for every 100 steps.

- Logging_hook

```
logging_hook = tf.train.LoggingTensorHook(
    tensors={'step': model.global_step,
             'loss': model.cost,
             'precision': precision},
    every_n_iter=100)
```

Figure A.42. Code Snippet: logging hook

- Prints logs after every 100 iterations.

A.4.5. Training from Scratch and/or Transfer Learning

A.4.5.1. Training

Open the *run* script and edit parameters as required.

```
python resnet_main.py \
    --train_data_path=/home/dataset/training/data/tfrecords/ \
    --log_root=./logs/train \
    --train_dir=./logs/train \
    --dataset='signlang' \
    --image_size=64 \
    --num_gpus=1 \
    --mode=train
```

Figure A.43. Predictive Maintenance – Run Script

To start training run the run script as mentioned below.

```
$ ./run
```

```
INFO:tensorflow:Graph was finalized.
I0421 12:06:05.044778 140199668410176 monitored_session.py:240] Graph was finalized.
INFO:tensorflow:Running local_init_op.
I0421 12:06:05.397494 140199668410176 session_manager.py:500] Running local_init_op.
INFO:tensorflow:Done running local_init_op.
I0421 12:06:05.415019 140199668410176 session_manager.py:502] Done running local_init_op.
I0421 12:06:06.224065 140199668410176 basic_session_run_hooks.py:606] Saving checkpoints for 0 into ./logs/train/model.ckpt.
INFO:tensorflow:loss = 2.7537637, precision = 0.03125, step = 0
I0421 12:06:07.673088 140199668410176 basic_session_run_hooks.py:262] loss = 2.7537637, precision = 0.03125, step = 0
INFO:tensorflow:loss = 8.1982155, precision = 0.984375, step = 34 (12.621 sec)
I0421 12:06:20.293941 140199668410176 basic_session_run_hooks.py:260] loss = 8.1982155, precision = 0.984375, step = 34 (12.621 sec)
INFO:tensorflow:loss = 8.56728, precision = 1.0, step = 67 (10.728 sec)
I0421 12:06:31.022384 140199668410176 basic_session_run_hooks.py:260] loss = 8.56728, precision = 1.0, step = 67 (10.728 sec)
INFO:tensorflow:global_step/sec: 2.78888
I0421 12:06:43.529809 140199668410176 basic_session_run_hooks.py:692] global_step/sec: 2.78888
INFO:tensorflow:loss = 8.464728, precision = 1.0, step = 100 (12.649 sec)
I0421 12:06:43.671326 140199668410176 basic_session_run_hooks.py:260] loss = 8.464728, precision = 1.0, step = 100 (12.649 sec)
INFO:tensorflow:loss = 8.351438, precision = 1.0, step = 134 (9.906 sec)
I0421 12:06:53.577040 140199668410176 basic_session_run_hooks.py:260] loss = 8.351438, precision = 1.0, step = 134 (9.906 sec)
```

Figure A.44. Predictive Maintenance – Trigger Training

A.4.5.2. Transfer Learning

```
INFO:tensorflow:Create CheckpointSaverHook.
I0421 12:27:24.113183 139632644269888 basic_session_run_hooks.py:541] Create CheckpointSaverHook.
INFO:tensorflow:Restoring parameters from ./logs/train/model.ckpt-4000
I0421 12:27:24.581049 139632644269888 saver.py:1280] Restoring parameters from ./logs/train/model.ckpt-4000
INFO:tensorflow:Saving checkpoints for 4000 into ./logs/train/model.ckpt.
I0421 12:27:25.606637 139632644269888 basic_session_run_hooks.py:606] Saving checkpoints for 4000 into ./logs/train/model.ckpt.
INFO:tensorflow:loss = 1.7730277, precision = 1.0, step = 4000
I0421 12:27:26.982490 139632644269888 basic_session_run_hooks.py:262] loss = 1.7730277, precision = 1.0, step = 4000
INFO:tensorflow:loss = 1.7497265, precision = 1.0, step = 4034 (11.972 sec)
I0421 12:27:38.954078 139632644269888 basic_session_run_hooks.py:260] loss = 1.7497265, precision = 1.0, step = 4034 (11.972 sec)
INFO:tensorflow:loss = 1.7260615, precision = 1.0, step = 4067 (9.745 sec)
I0421 12:27:48.698793 139632644269888 basic_session_run_hooks.py:260] loss = 1.7260615, precision = 1.0, step = 4067 (9.745 sec)
INFO:tensorflow:global_step/sec: 3.12938
I0421 12:27:58.937695 139632644269888 basic_session_run_hooks.py:692] global_step/sec: 3.12938
INFO:tensorflow:loss = 1.7033625, precision = 1.0, step = 4100 (10.303 sec)
I0421 12:27:59.001640 139632644269888 basic_session_run_hooks.py:260] loss = 1.7033625, precision = 1.0, step = 4100 (10.303 sec)
INFO:tensorflow:loss = 1.6810057, precision = 1.0, step = 4134 (9.778 sec)
I0421 12:28:08.779979 139632644269888 basic_session_run_hooks.py:260] loss = 1.6810057, precision = 1.0, step = 4134 (9.778 sec)
INFO:tensorflow:loss = 1.6583471, precision = 1.0, step = 4167 (10.514 sec)
I0421 12:28:19.294208 139632644269888 basic_session_run_hooks.py:260] loss = 1.6583471, precision = 1.0, step = 4167 (10.514 sec)
```

Figure A.45. Predictive Maintenance – Trigger Training with Transfer Learning

- To restore checkpoints, no additional action is required. Run the same command again with the same log directory. if the checkpoints are present in log path where it is be restored and continue training from that step.

A.4.5.3. Training Status

- Training status can be checked in logs by observing different terminologies like loss, precision and confusion matrix.

```
I0707 12:36:19.063314 139684916700992 basic_session_run_hooks.py:260] loss = 0.18542665, precision = 0.984375, step = 8500 (5.558 sec)
INFO:tensorflow:loss = 0.20943533, precision = 0.9609375, step = 8550 (5.665 sec)
I0707 12:36:24.728753 139684916700992 basic_session_run_hooks.py:260] loss = 0.20943533, precision = 0.9609375, step = 8550 (5.665 sec)
INFO:tensorflow:global_step/sec: 8.87325
I0707 12:36:30.285727 139684916700992 basic_session_run_hooks.py:692] global_step/sec: 8.87325
INFO:tensorflow:loss = 0.22918972, precision = 0.96875, step = 8600 (5.601 sec)
I0707 12:36:30.329452 139684916700992 basic_session_run_hooks.py:260] loss = 0.22918972, precision = 0.96875, step = 8600 (5.601 sec)
INFO:tensorflow:loss = 0.2660838, precision = 0.96875, step = 8650 (5.712 sec)
I0707 12:36:36.041846 139684916700992 basic_session_run_hooks.py:260] loss = 0.2660838, precision = 0.96875, step = 8650 (5.712 sec)
INFO:tensorflow:global_step/sec: 8.83564
I0707 12:36:41.603530 139684916700992 basic_session_run_hooks.py:692] global_step/sec: 8.83564
INFO:tensorflow:loss = 0.2278277, precision = 0.9609375, step = 8700 (5.610 sec)
I0707 12:36:41.652254 139684916700992 basic_session_run_hooks.py:260] loss = 0.2278277, precision = 0.9609375, step = 8700 (5.610 sec)
```

Figure A.46. Predictive Maintenance – Training Logs

```
Confusion_Matrix :
 [[ 14726      0      0]
 [     0 113274      0]
 [     0      0      0]]
```

Figure A.47. Predictive Maintenance – Confusion Matrix

- You can use Tensorboard utility for checking training status.
  - Start Tensorboard by below command:

```
$ tensorboard –logdir=<log directory of training>
```

```
$ tensorboard --logdir logs/train/
TensorBoard 1.15.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

Figure A.48. Tensorboard – Launch

  - This command provides the link, which needs to be copied and open in any browser such as Chrome, Firefox, and others or right-click on the link and click **Open Link**.

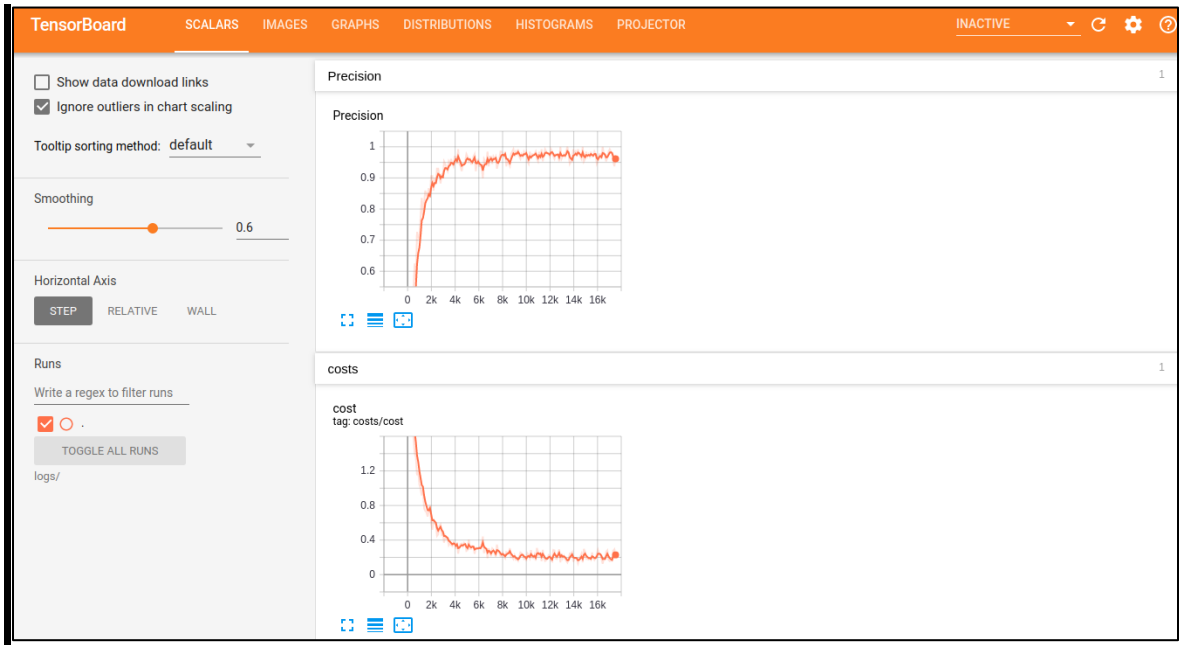

Figure A.49. Tensorboard – Link Default Output in Browser

- Similarly, other graphs can be investigated from the available list.
- Check if the *checkpoint*, *data*, *meta* and *index* files are created at the log directory. These files are used for creating the frozen file (*.pb).

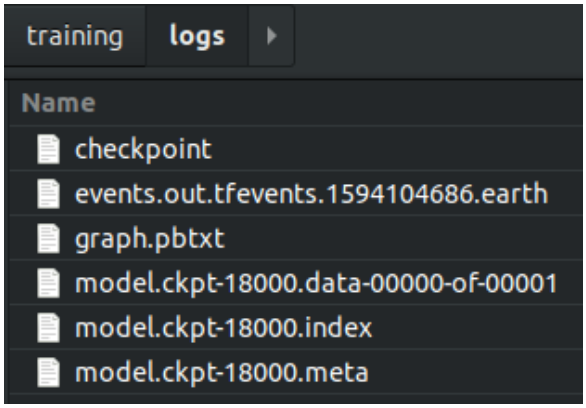

Figure A.50. Checkpoint Storage Directory Structure

A.5. Creating Frozen File

This section describes the procedure for freezing the model, which is aligned with the Lattice sensAI tool. Perform the steps below to generate the frozen protobuf file:

A.5.1. Generating .pbtxt File for Inference

Once the training is completed run below command to generate inference .pbtxt file.
Note: Do not modify *config.sh* after training.

```
$ python resnet_main.py --train_data_path=<TFRecord_root_path> --
log_root=<Logging_Checkpoint_Path> --train_dir=<tensorboard_summary_path> --
dataset='signlang' --image_size=64 --num_gpus=<num_GPUs> --mode=freeze
```

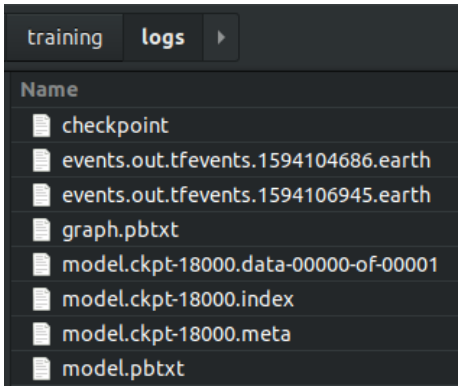

Figure A.51. Generated '.pbtxt' for Inference

It generates the .pbtxt file for inference under the train log directory.

A.5.2. Generating the Frozen (.pb) File

```
$ python genpb.py --ckpt_dir <COMPLETE_PATH_TO_LOG_DIRECTORY>
```

```
inputShape shape [None, None, None, None]
inputShape shapes [None, None, None, None]
output_shapes of input Node [None, None, None, None]
 **TensorFlow**: can not locate input shape information at: random_shuffle_queue_DequeueMany
node to modify name: "random_shuffle_queue_DequeueMany"
op: "Placeholder"
attr {
  key: "dtype"
  value {
    type: DT_FLOAT
  }
}

--Name of the node - random_shuffle_queue_DequeueMany shape set to  random_shuffle_queue_DequeueMany [1, 64, 64, 1]
node after modify name: "random_shuffle_queue_DequeueMany"
op: "Placeholder"
attr {
  key: "dtype"
  value {
    type: DT_FLOAT
  }
}
```

Figure A.0.52. Run genpb.py to Generate Inference .pb

- *genpb.py* uses *.pbtxt* generated by procedure in the Generating .pbtxt File for Inference section and latest checkpoint in train directory to generate frozen .pb file.
- Once the genpb.py is executed successfully, the <ckpt-prefix>_frozenforinference.pb becomes available in the log directory as shown in below figure

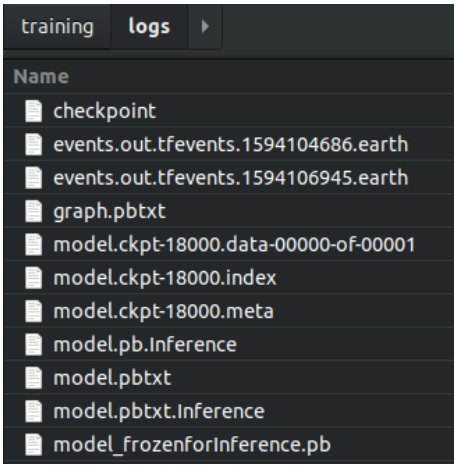

Figure A.53. Frozen Inference .pb Output

A.6. TensorFlow Lite Conversion and Evaluation

This section contains information for converting frozen pb to TensorFlow Lite model, quantize the model and evaluate on test dataset.

Note: It is recommended to use *Tensorflow 2.2.0 (CPU Only)* instead *Tensorflow 1.15.0* In TensorFlow Lite conversion flow. Use Environment created from the Creating the TensorFlow Lite Conversion Environment section**.**

A.6.1. Converting Frozen Model to TensorFlow Lite

You can find the *gen_tflite_and_quant.py* under training code, which converts the frozen model to TensorFlow Lite and also quantize it with INT8 quantization.

```
$ python gen_tflite_and_quant.py --input_path <sample images path> --tflite_path
<output tflite path> --pb <frozen pb file>
```

The following are the argument information:

- --input_path: sample images that are used for quantization.
- --tflite_path: (default motor-model.tflite) output tflite path
- --pb: Frozen pb path

The command saves the TensorFlow Lite at given path.

A.6.2. Evaluating TensorFlow Lite Model

```
$ python evaluate_tflite.py --dataset_path <dataset_path> --tflite_path <tflite
path>
```

The following are the argument information:

- --dataset_path: Test set path. Note that the labels should be (0, 1) for predictive maintenance.
- --tflite_path: tflite model path

The command shows accuracy on both classes.

A.6.3. Converting TensorFlow Lite To C-Array

```
$ xxd -i your-tflite-model-path.tflite > out_c_array.cc
```

The command generates the *c* array at the path you provided.

Refer to Automate Stack 3.1 Demo User Guide (FPGA-UG-02164) for detailed instructions on compiling the code, installing the client-end application, automate stack 3.1 bit file and binary, programming the Automate Stack on SPI Flash memory, troubleshooting the main system board, and debugging using Docklight, OPCUA Modeler, and CSV file.

Appendix B. Setting Up the Wireshark Tool

Note: To download the wireshark tool, go to https://www.wireshark.org/download.html.

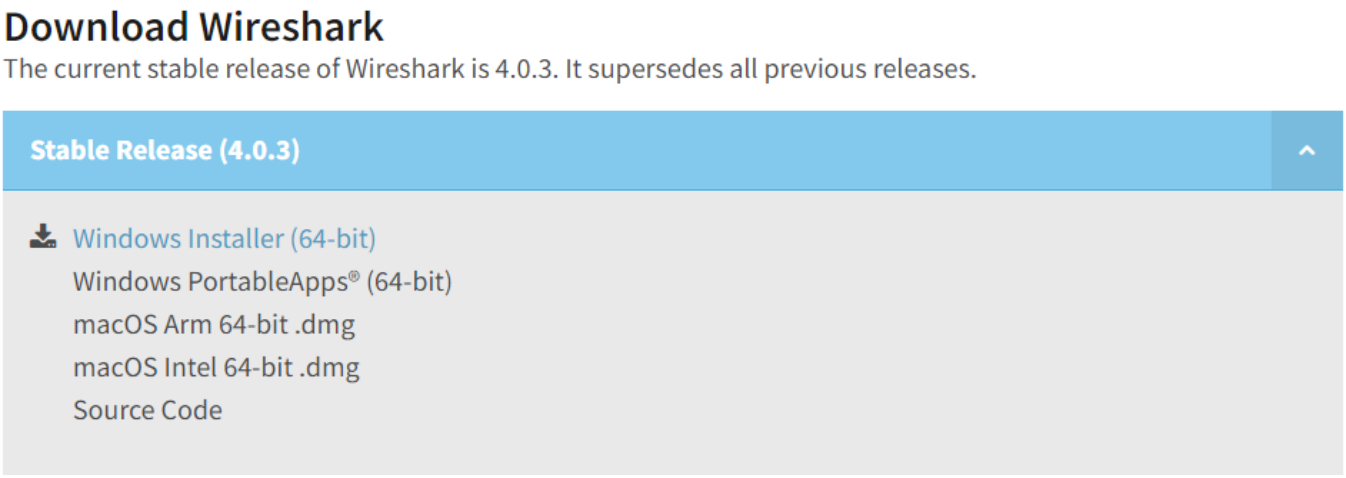

Figure B.1. Wireshark Downloadable Link

To set up the Wireshark tool, perform the following steps:

1. Open the Wireshark tool and select the network (Ethernet).
2. Click on the Ethernet network.

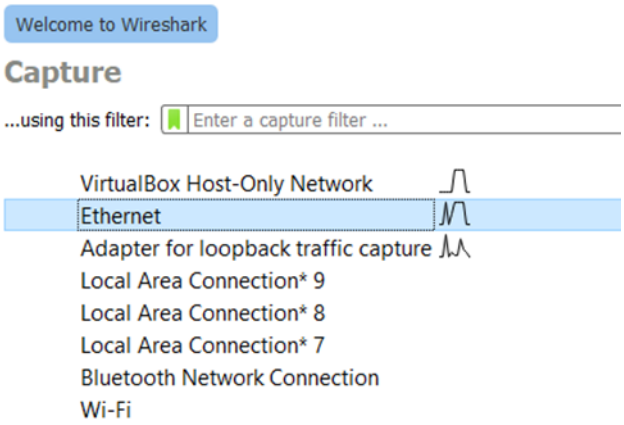

Figure B.2. Wireshark Tool – Ethernet selection

3. Click on the **Run** ( ) button.
4. Check the UDP message use port filter (udp.port == 1486) on the top bar.

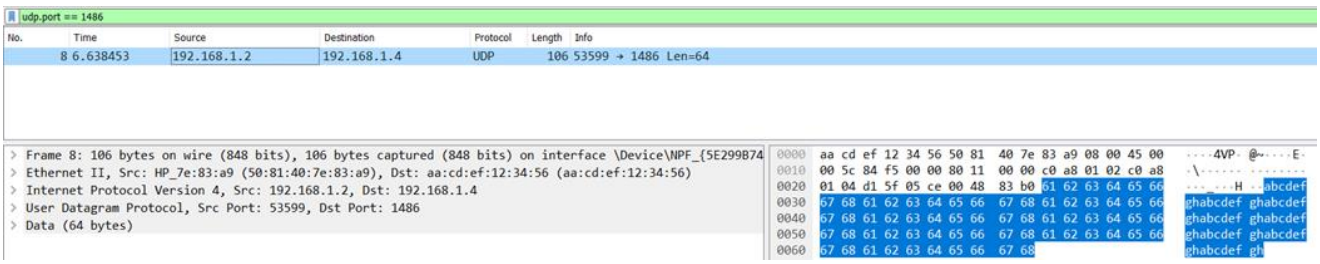

Figure B.3. Wireshark Tool – Write udp.port == 1486

5. Check both the source and destination IP.

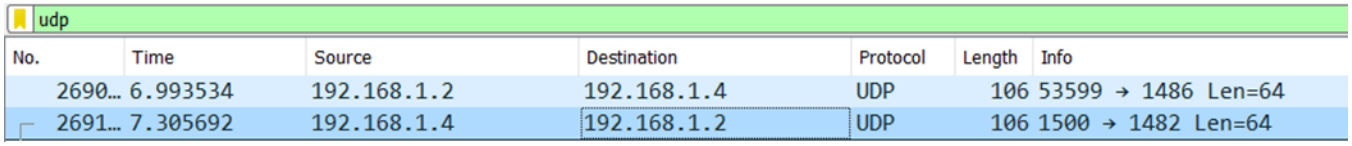

Figure B.4. Source and Destination UDP Packet

6. Click the UDP packet.

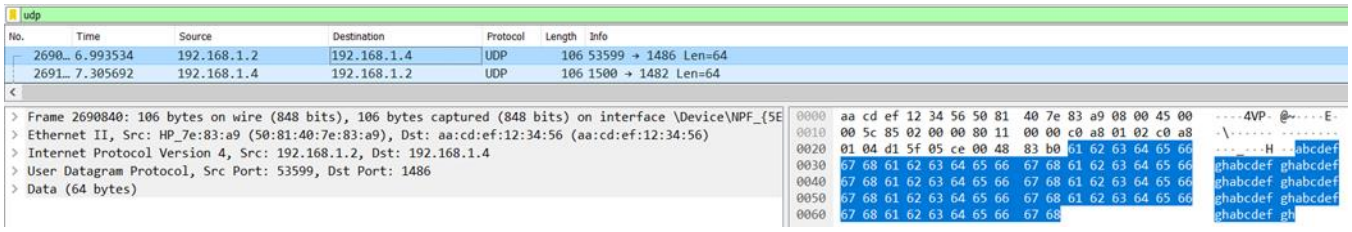

Figure B.5. Wireshark Tool – First UDP Packet

Appendix C. Generating Automate Stack 3.1 Propel Patch and Bitstream

Lattice Automate solution stack project files can be downloaded by installing the Automate propel patch from the Lattice Automate web page.

C.1. Installing the Propel SDK 2023.2

To install the Propel SDK, perform the following steps:

1. Double-click on the application to install and click **Yes** on the pop-out window.

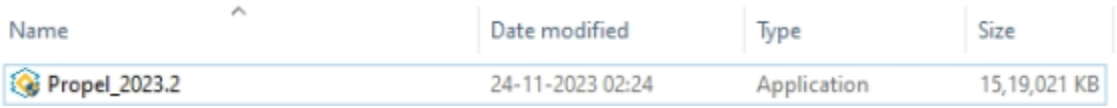

Figure C.1. Propel Application

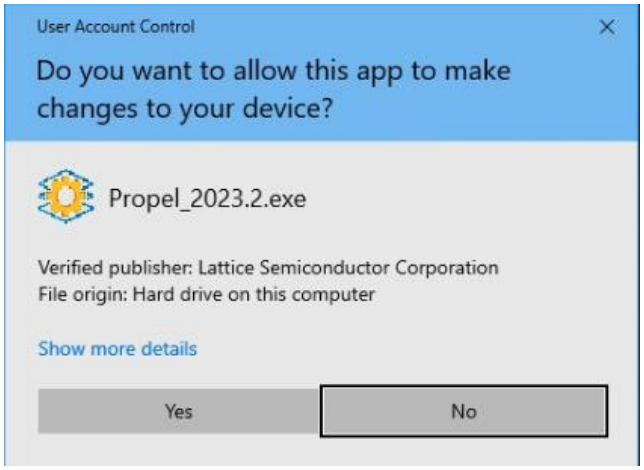

Figure C.2. Allow Permission

2. Click **Next** as shown in Figure C.3 to Figure C.5.

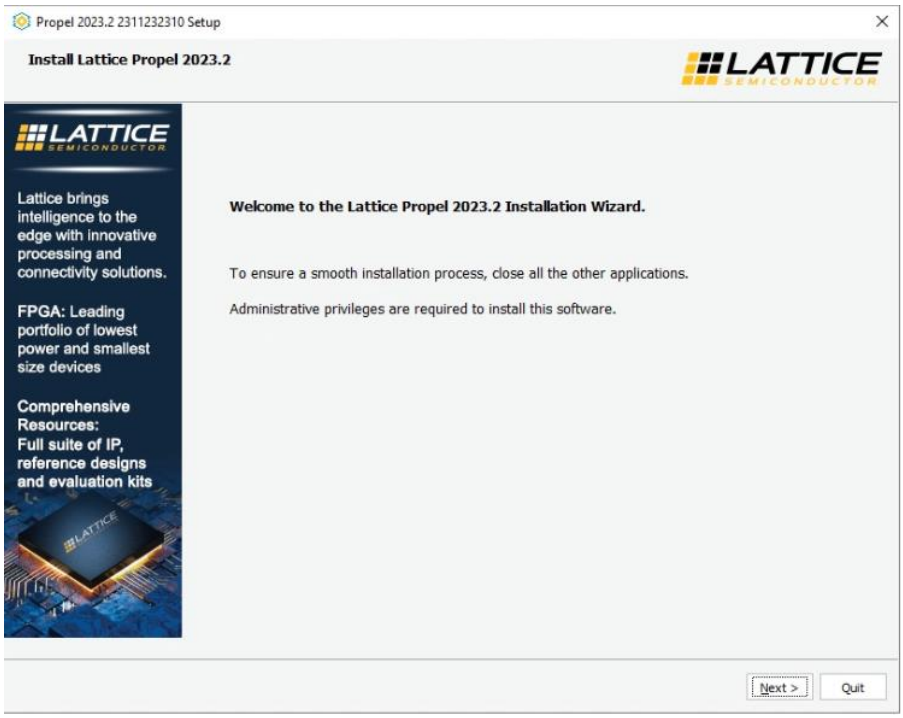

Figure C.3. Lattice Propel 2023. 2 Installation Wizard

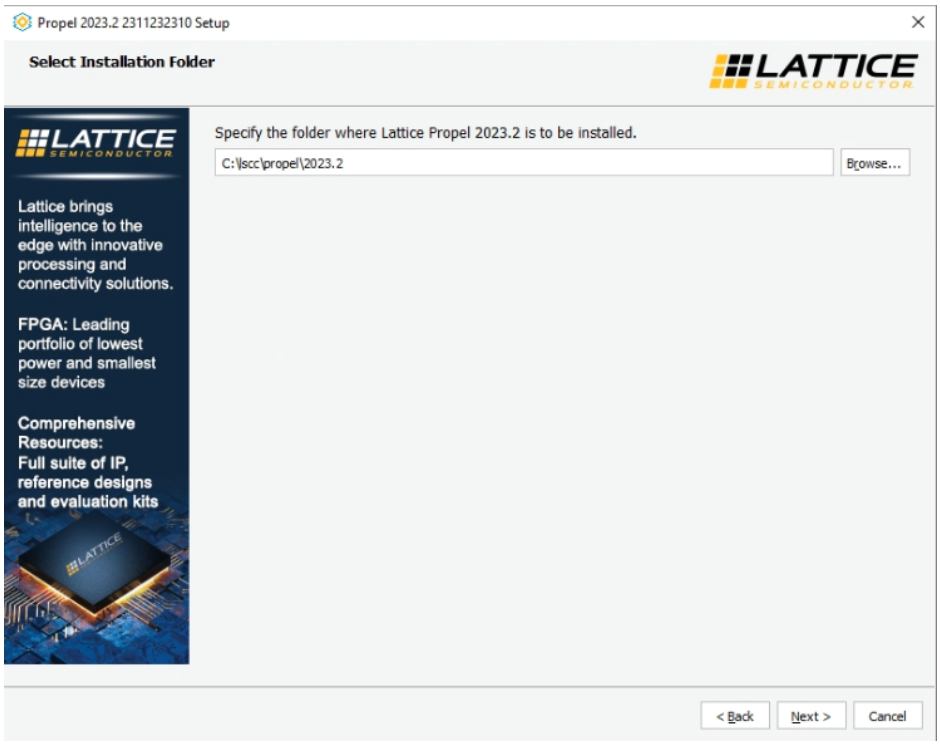

Figure C.4. Select Installation Folder

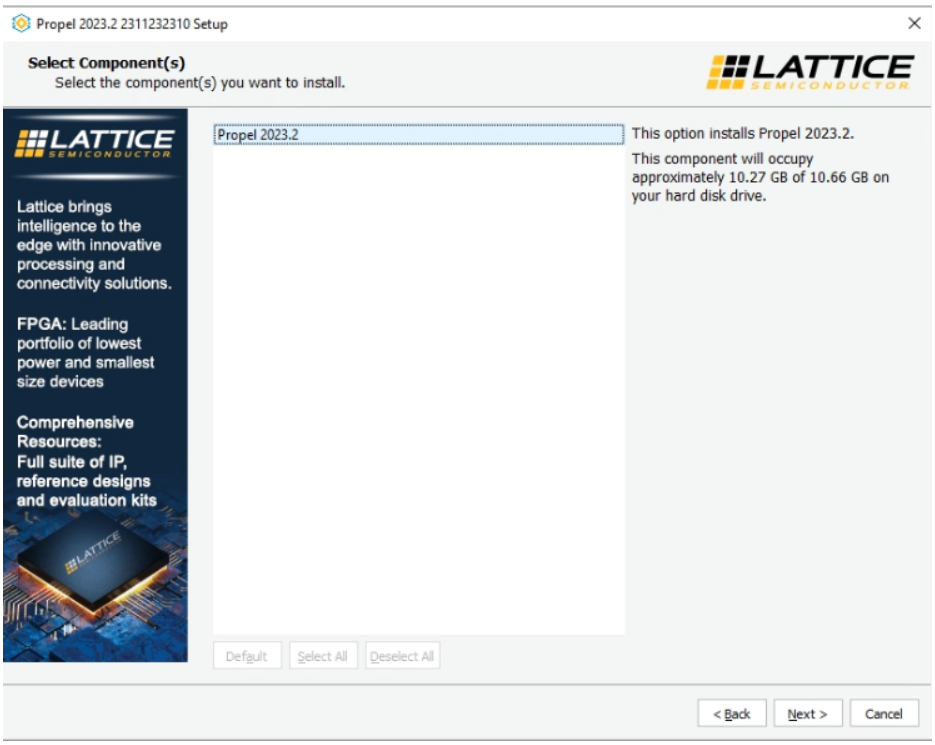

Figure C.5. Install Components

3. Select **I accept the license** and click **Next** as shown in Figure C.6 to Figure C.7.

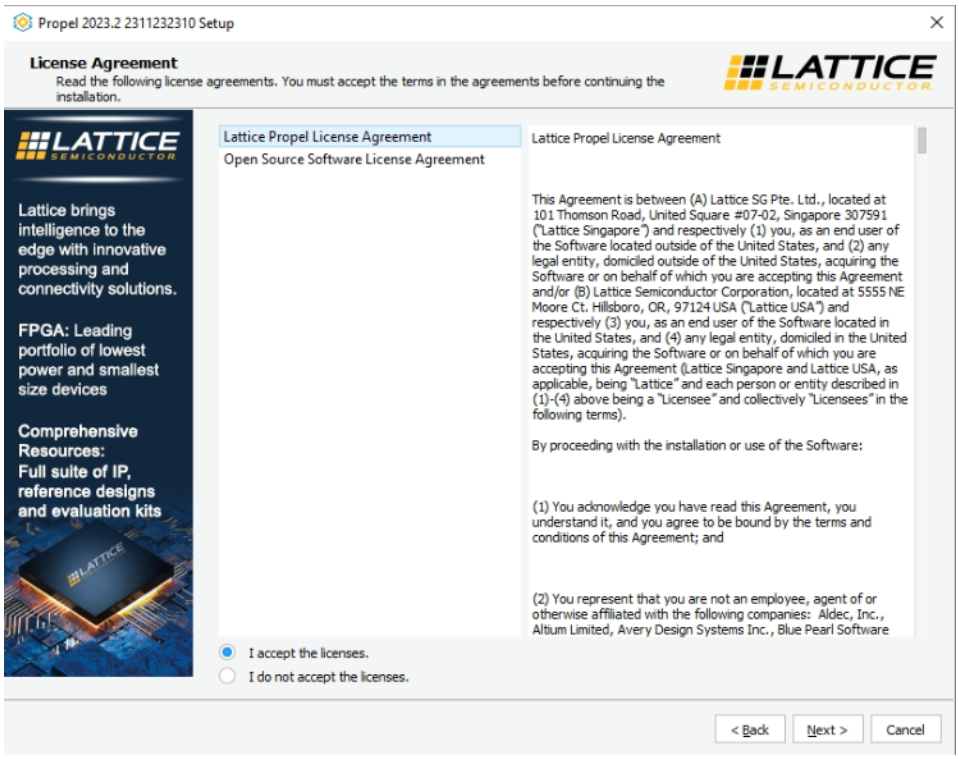

Figure C.6. Accept the License

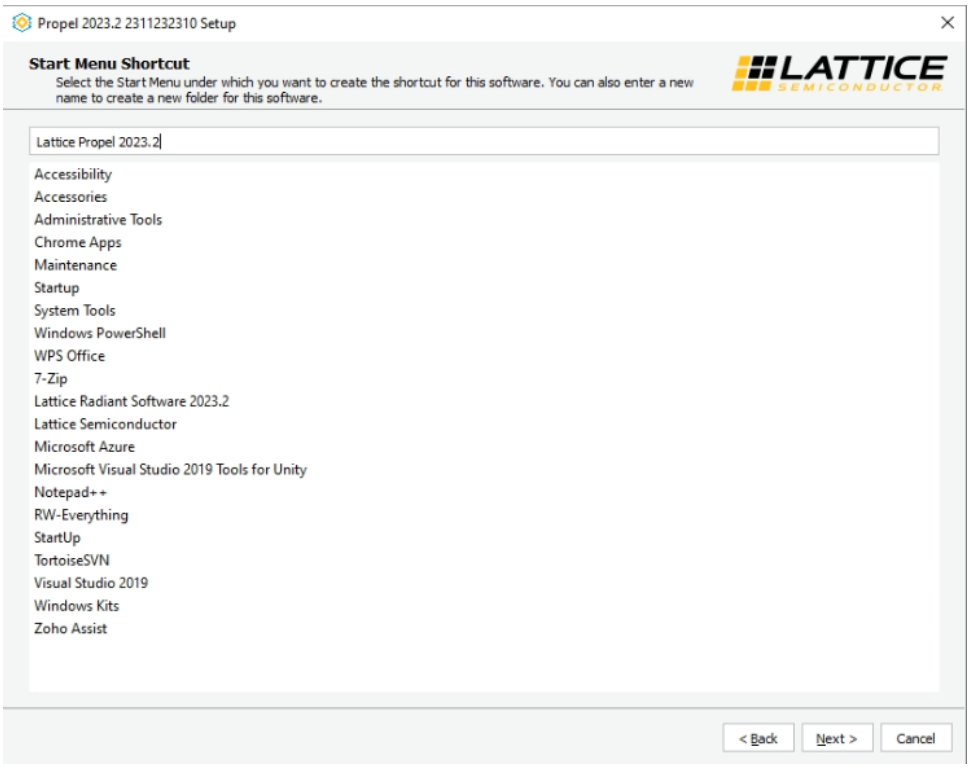

Figure C.7. Start Menu Shortcut

4. Click **Install**.

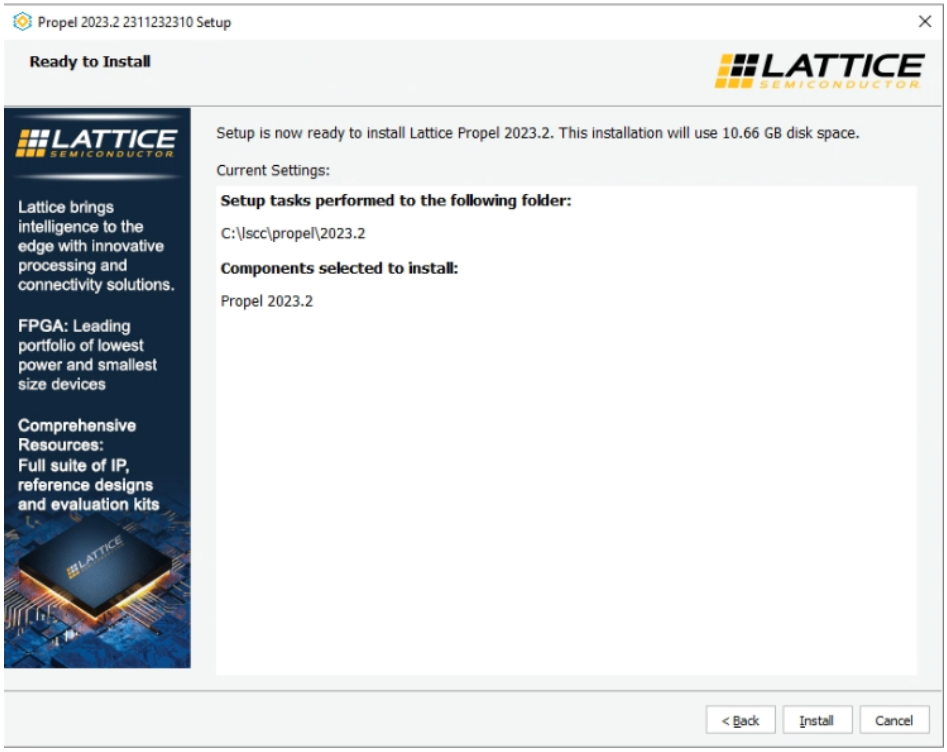

Figure C.8. Install the Propel SDK Application

5. Wait for the installation process to reach 100%.

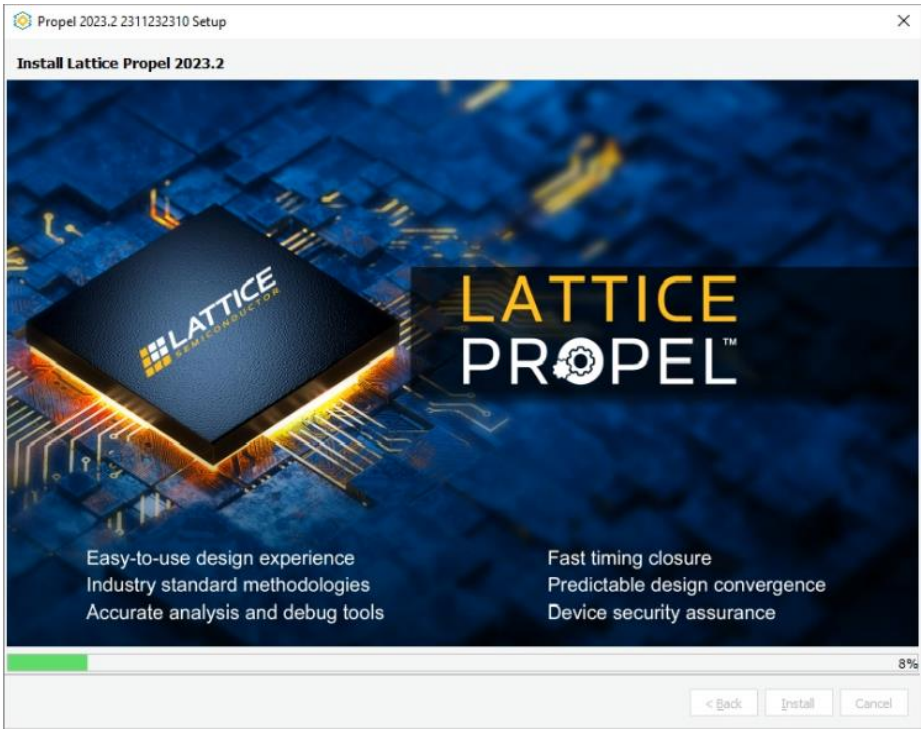

Figure C.9. Installation Process

6. Click **Finish** once installation is complete.

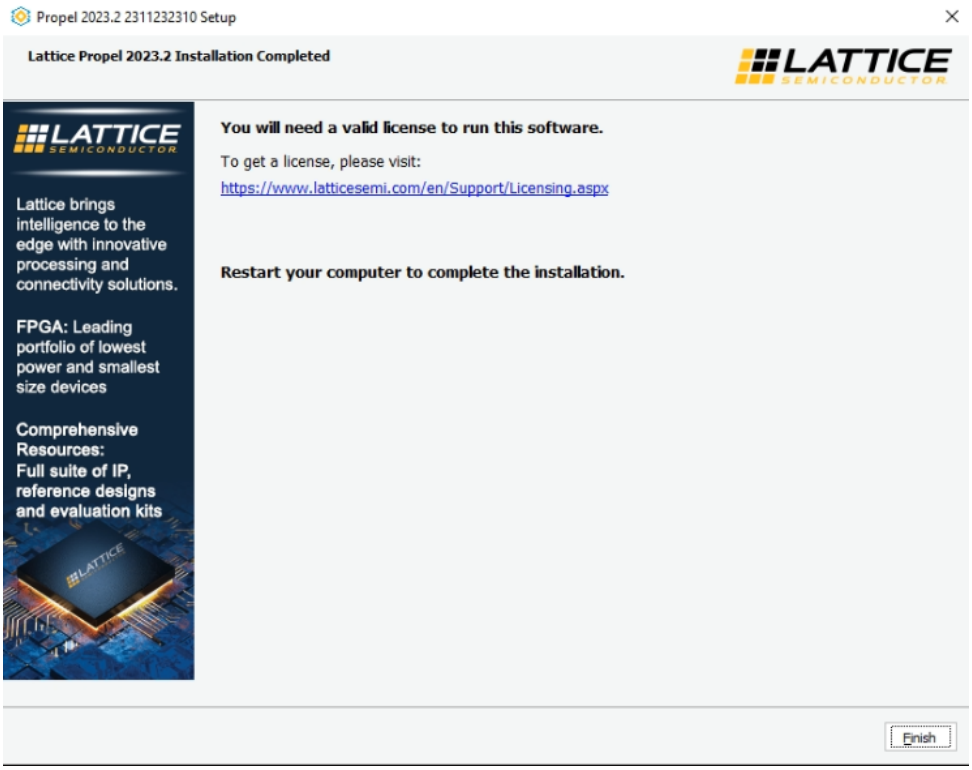

Figure C.10. Installation Complete

7. Paste the downloaded license to the path: **C:\lscc\propel\2023.2\license**.

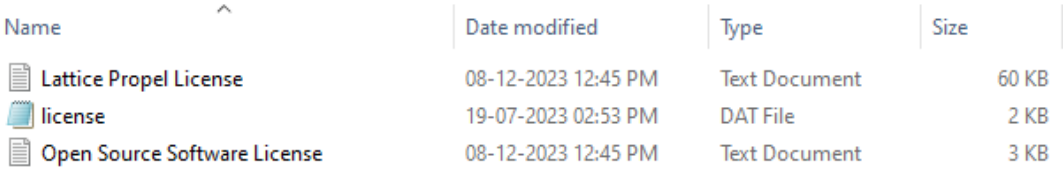

Figure C.11. License Path

8. Install the Lattice Automate 3.1 Propel patch. Follow steps as above to install Automate Propel patch.

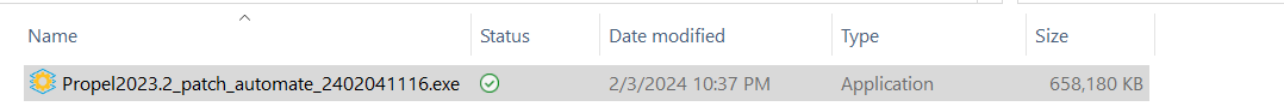

Figure C.12. Automate 3.1 Propel Patch

C.2. Generating the Binary

C.3.1. Primary Main System

To generate the binary in the primary main system, perform the steps below:

1. Double-click **Lattice Propel SDK 2023.2** to open the dialogue box as shown in Figure C.13.

Figure C.13. Propel 2023.2 Application

2. To select the workspace, browse to the template location or where your project is located: **\Main_System\Primary_MainSystem**. Click the **Launch** button to launch the workspace.

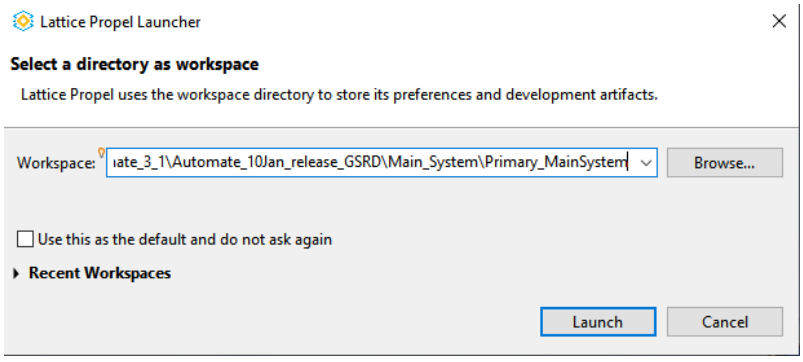

Figure C.14. Select Directory

3. Click **Import projects** or go to the **Import** from **file** to import firmware project template.

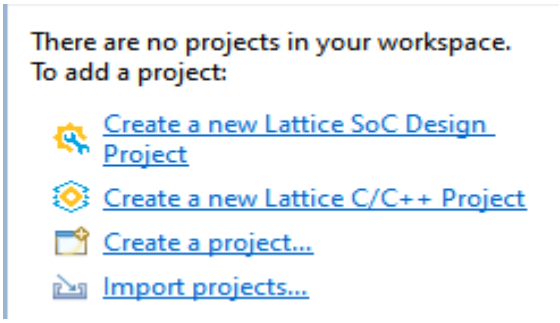

Figure C.15. Import Project

4. Select Existing Project in Workspace from General list and click on next as shown in Figure C.16.

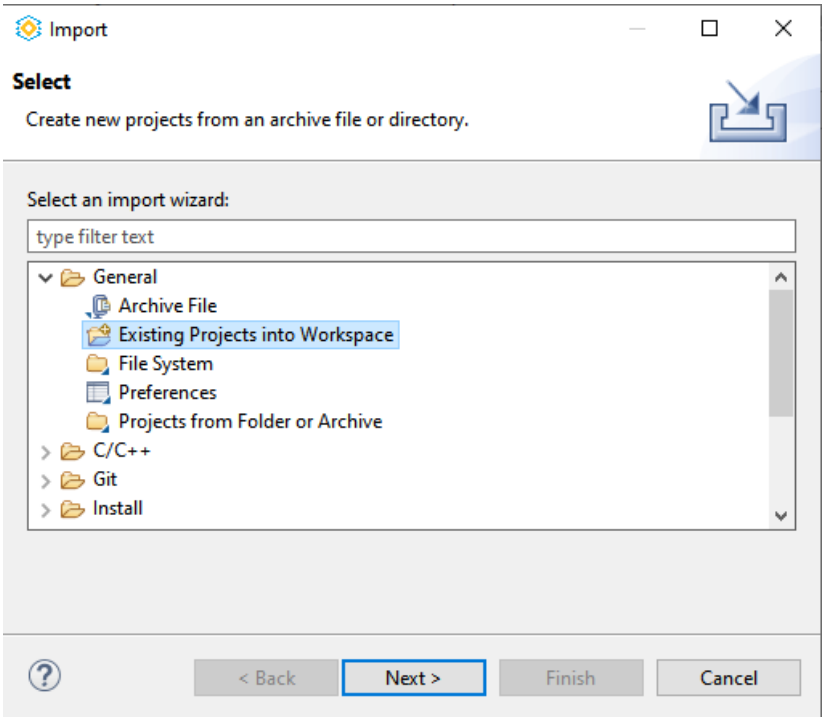

Figure C.16. Existing Project into Workspace

5. Select the root directory and browse template location.
6. Select the project as shown in Figure C.18: **\Main_System\Primary_MainSystem**.
7. Click **Finish**.

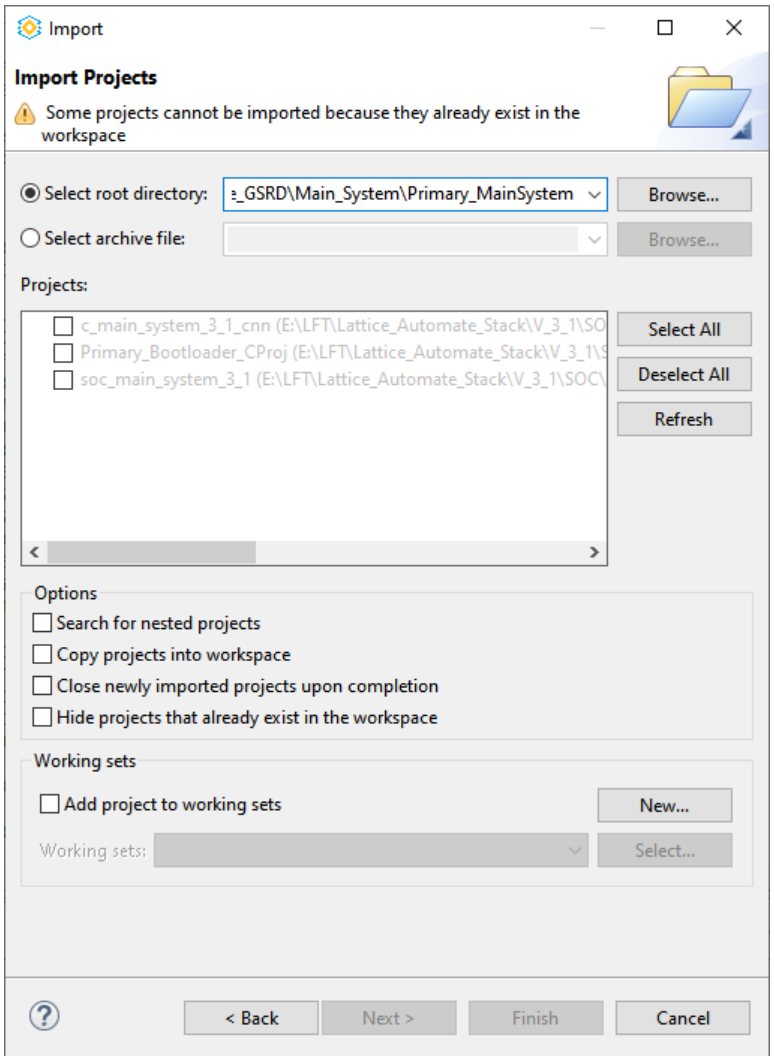

Figure C.17. Import Project

8. Right-click on the firmware project folder **c_main_system_3_1_cnn** and select **Properties**.

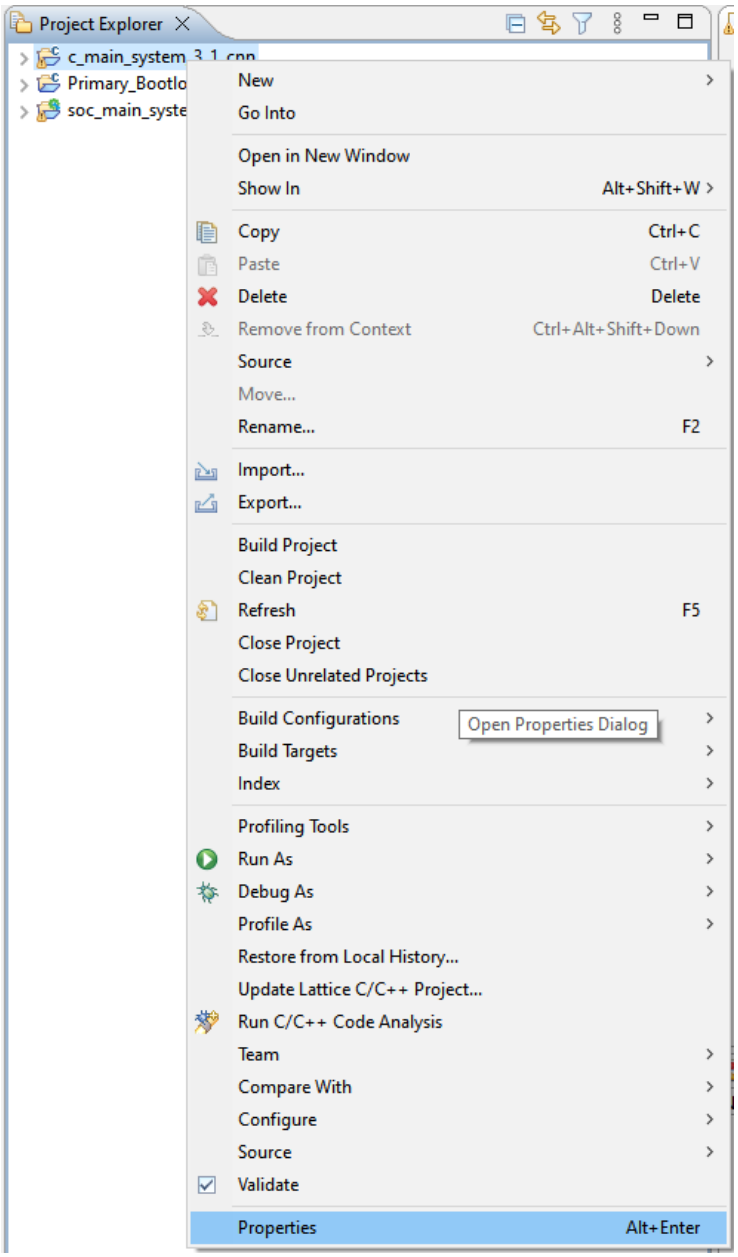

Figure C.18. Properties

9. Go to **C/C++ build > Settings** and click **Manage Configurations**.

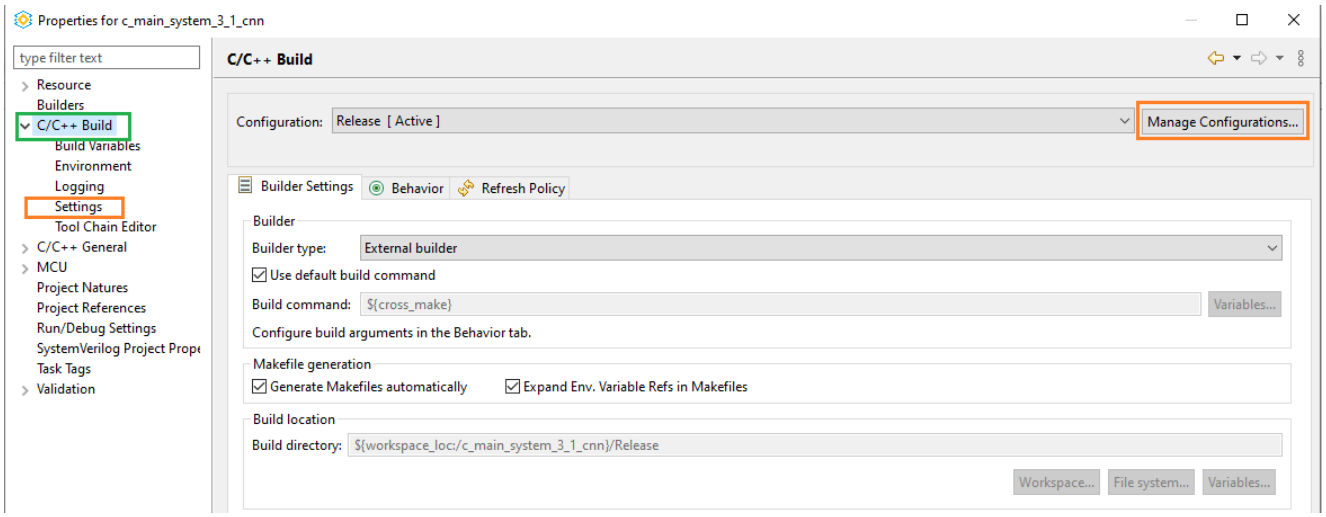

Figure C.19. C/C++ build settings

10. Select **Release** and apply **Set Active**. Click **OK**.

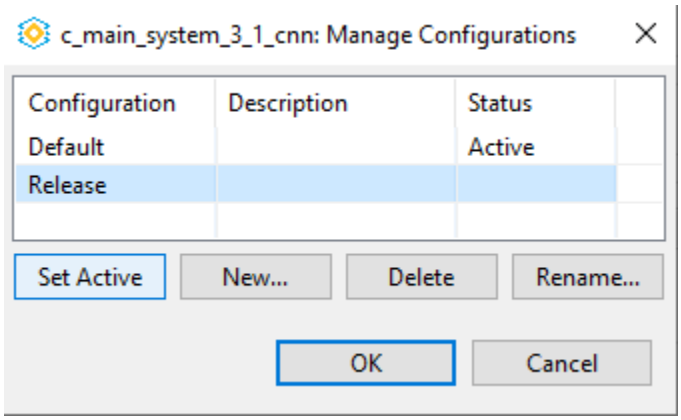

Figure C.20. Manage Configuration – Release: Set Active

11. Click **Apply** and close.

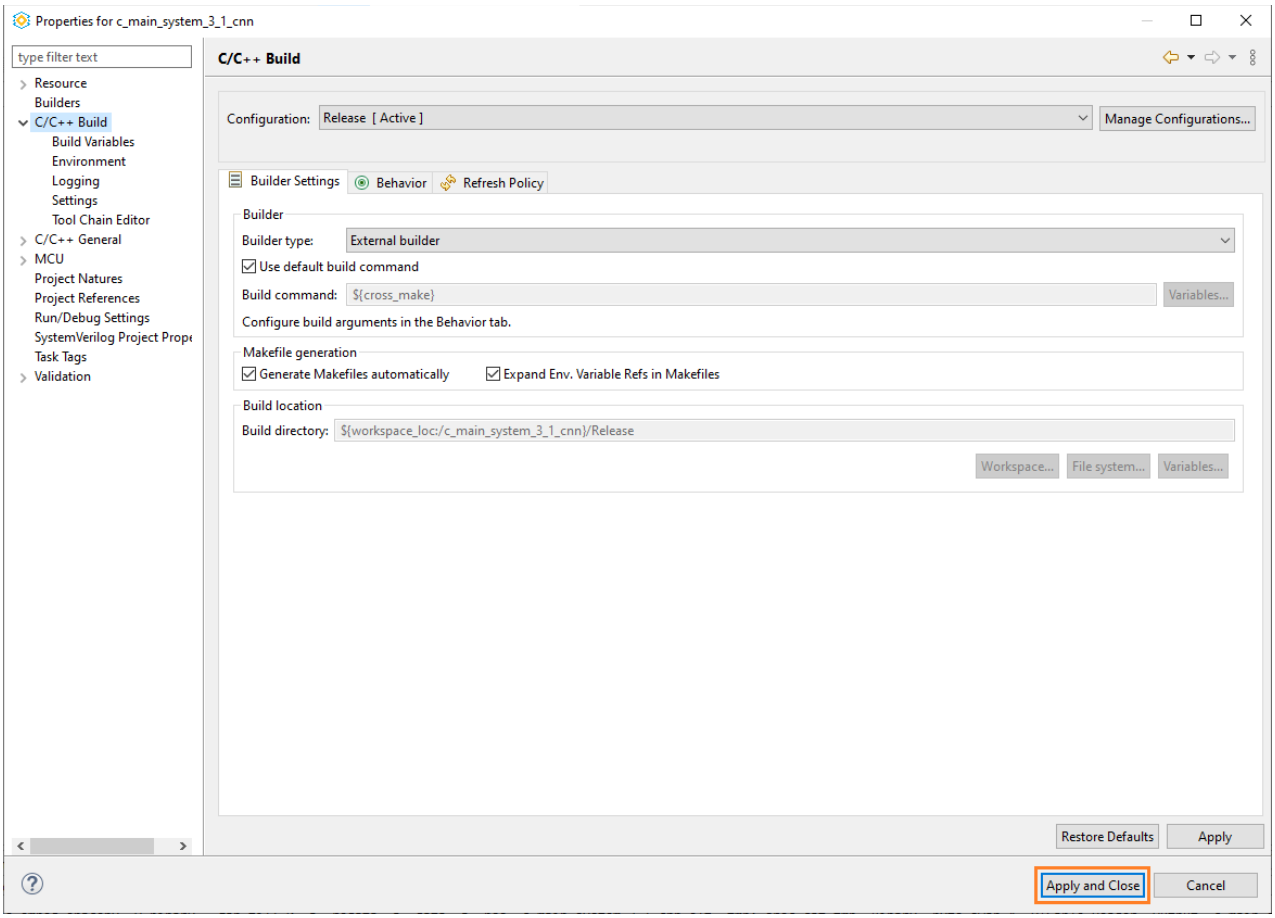

Figure C.21. Manage Configuration: Apply and Close

12. Right-click on the firmware project folder *c_main_system_3_1_cnn* and select the option as shown in Figure C.22 to clean the project before building.

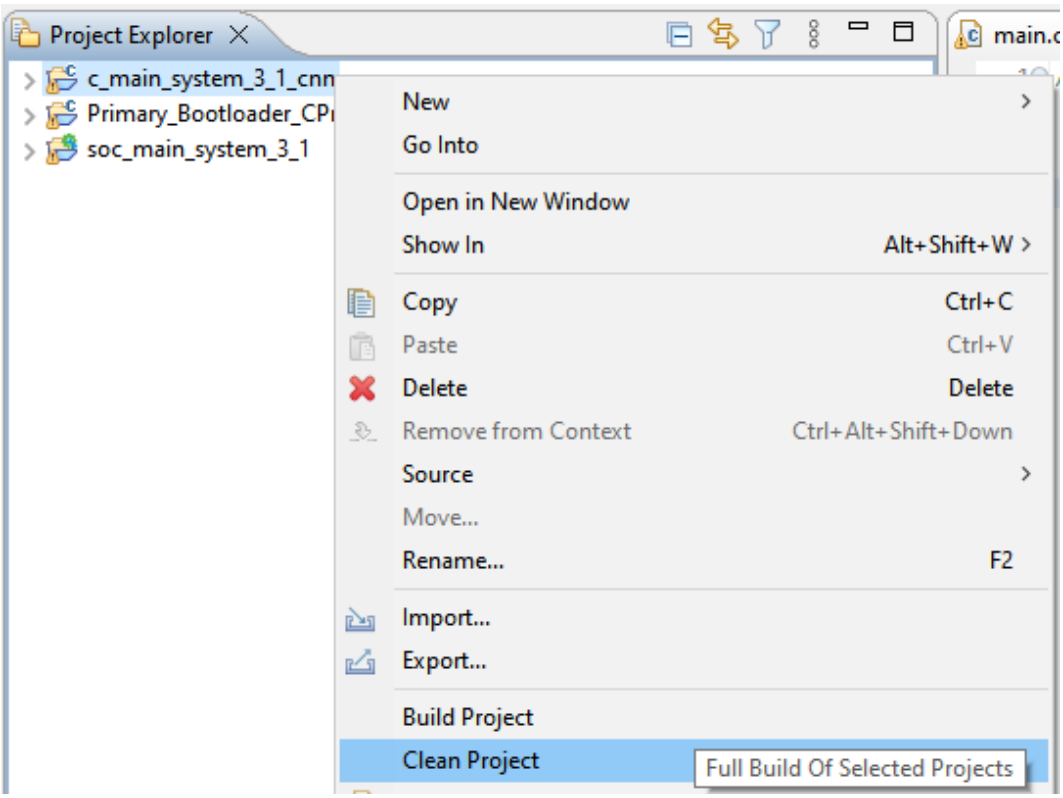

Figure C.22. Clean project Configurations

13. After selecting the option as shown in above fig. observe the console and wait for the process to complete to 100%. After completion, the message shown in Figure C.23 appears on the console.

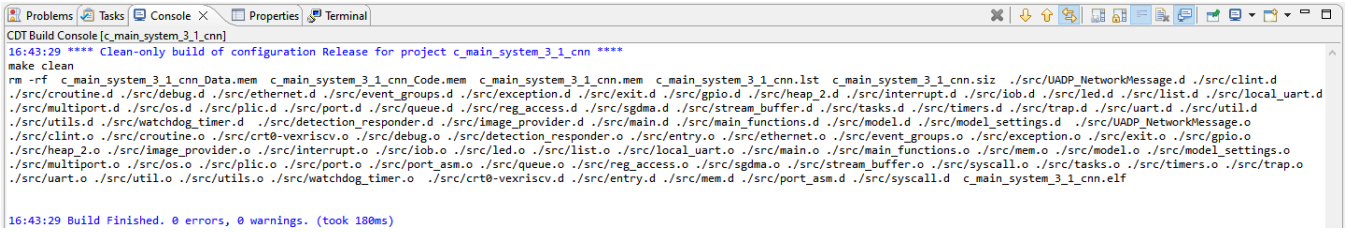

Figure C.23. Console

14. After cleaning, right click on the “c_main_system_3_1_cnn” and select the option as shown in Figure C.24 to build the project.

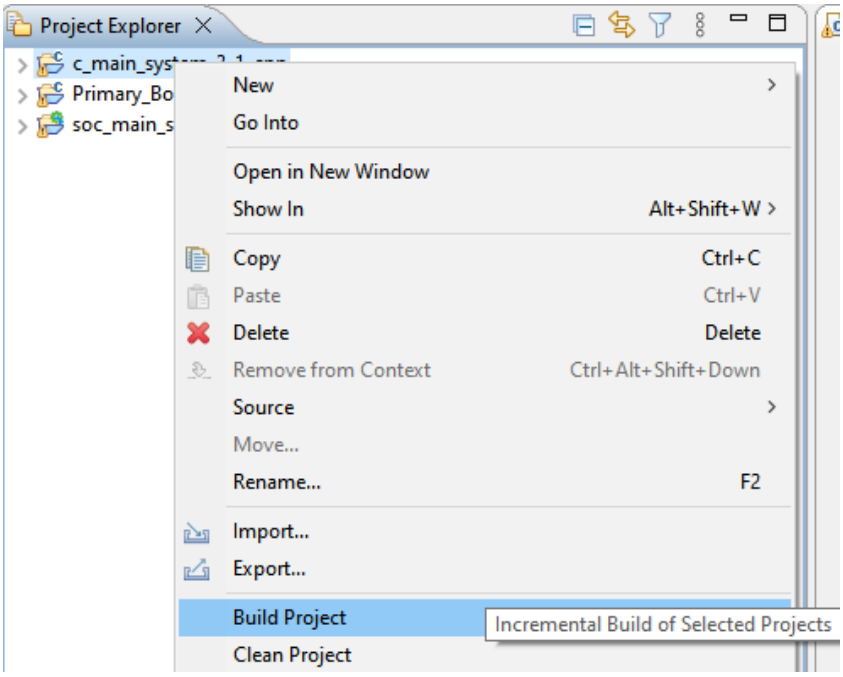

Figure C.24. Build Project

15. Wait for the process to complete to 100%. After completion, the message shown in Figure C.25 appears on the console.

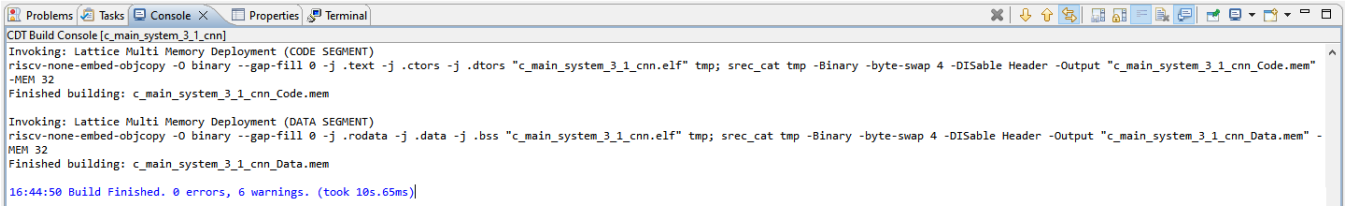

Figure C.25. Completing Process

16. Locate the binary file: \Main_System\Primary_MainSystem\c_main_system_3_1_cnn\Release.
17. Right-click on the bootloader project folder “Primary_Bootloader_Cproj” and select the option as shown in Figure C.26 to clean the project before building.

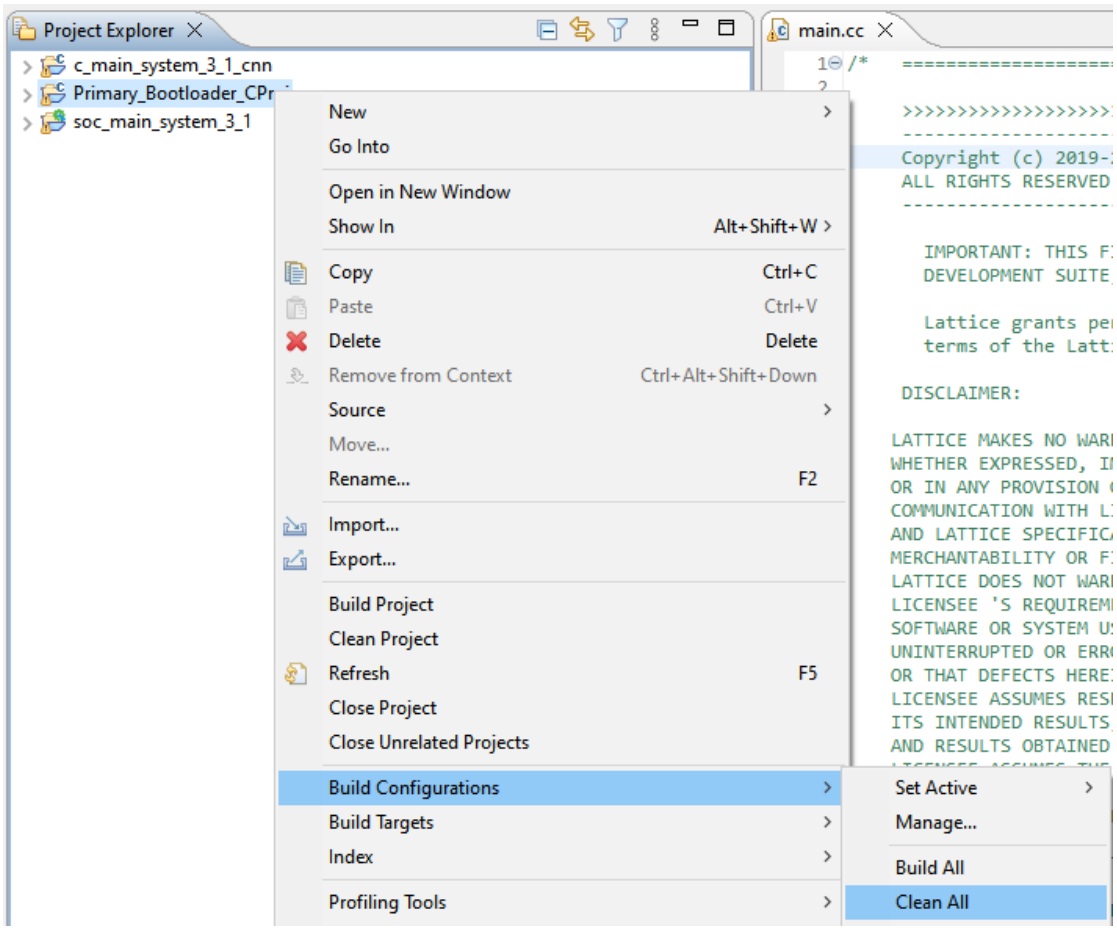

Figure C.26. Clean All Configurations

18. After selecting the option as shown in above fig. observe the console and wait for the process to complete to 100%. After completion, the message shown in Figure C.27 appears on the console.

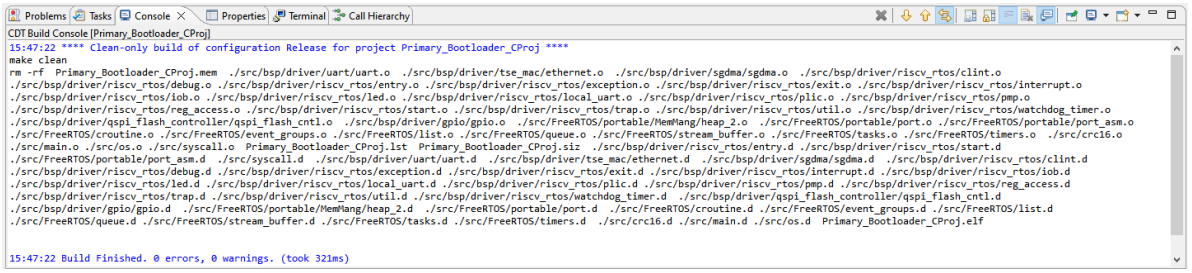

Figure C.27. Console

19. After cleaning, right-click on the *Primary_Bootloader_Cproj* and select the option as shown in Figure C.28 to build the project.

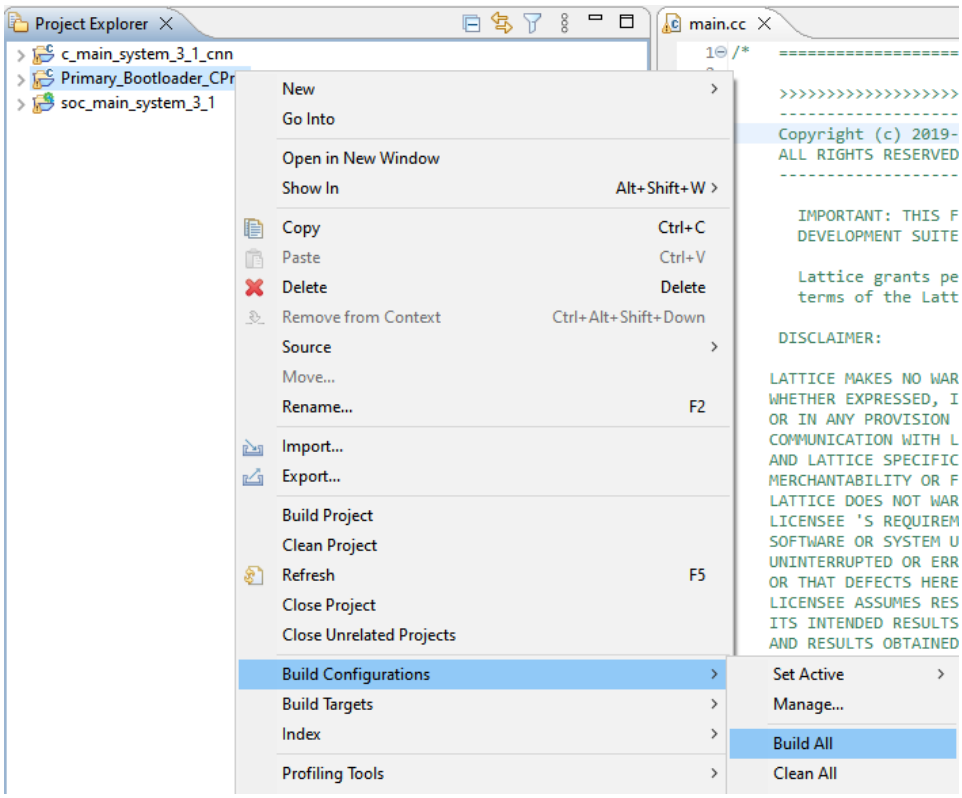

Figure C.28. Build All Configurations

20. Wait for the process to complete to 100%. After completion, the message shown in Figure C.29 appears on the console.

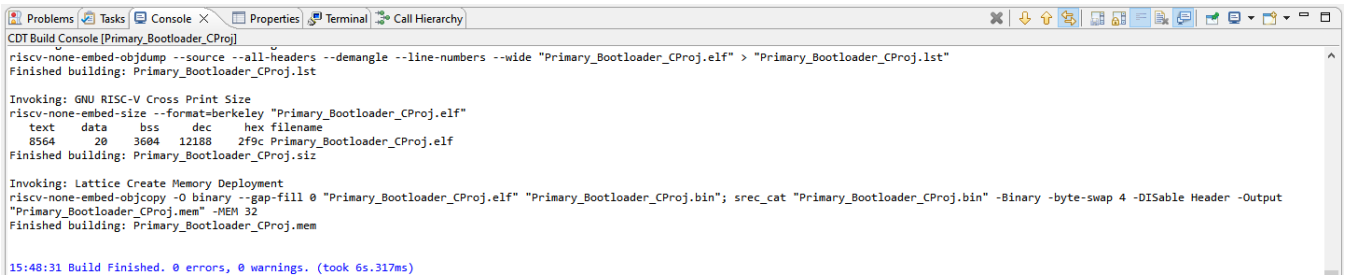

Figure C.29. Completing Process

21. To locate the binary file to below path: \MainSystem\Primary_MainSystem\Primary_Bootloader_CProj\Debug.

C.3.2. Golden Main System

To generate the binary in the golden main system, perform the steps below:

1. Double-click **Lattice Propel SDK 2023.2** to open the dialogue box.

Figure C.30. Propel 2023.2 Application

2. To select the workspace, browse to the template location or where your project is located: **\MainSystem\Golden_MainSystem**. Click the **Launch** button to launch the workspace.

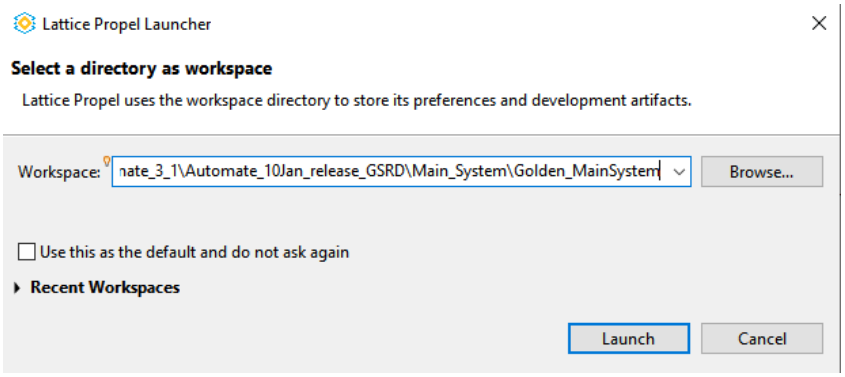

Figure C.31. Select Directory

3. Click **Import projects** or go to **Import > File** to import the firmware project template.

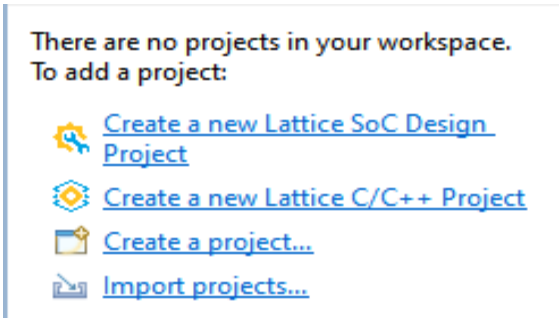

Figure C.32. Import Project

4. Select **Existing Project in Workspace** from the general list and click **Next** as shown in Figure C.33.

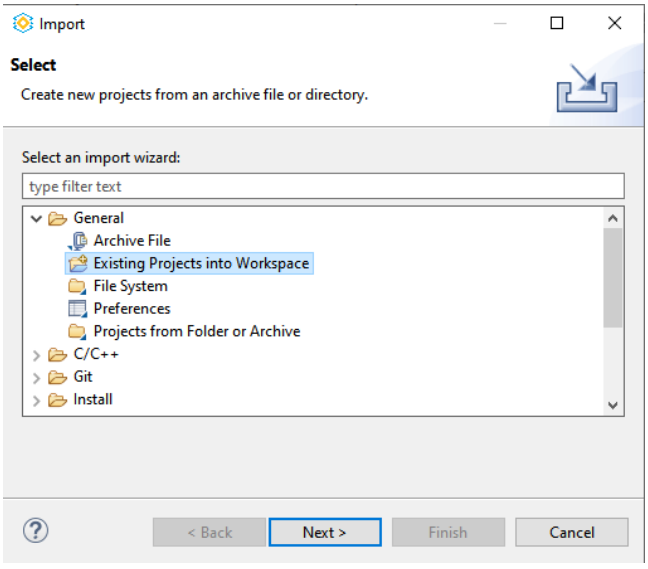

Figure C.33. Existing Project into Workspace

5. Select the root directory and browse template location.
6. Select the project as shown in Figure C.34: **\MainSystem\Golden_MainSystem**.
7. Click **Finish**.

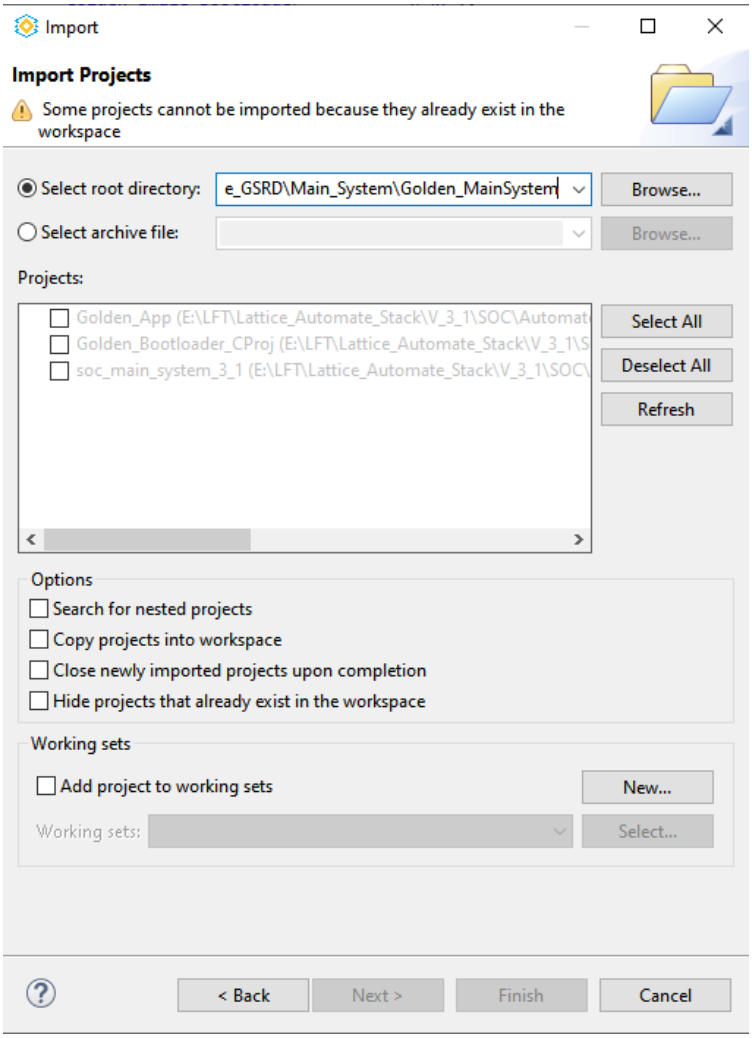

Figure C.34. Import Project

8. Right-click on the firmware project folder **Golden_App** and select the option as shown in Figure C.35 to clean the project before building.

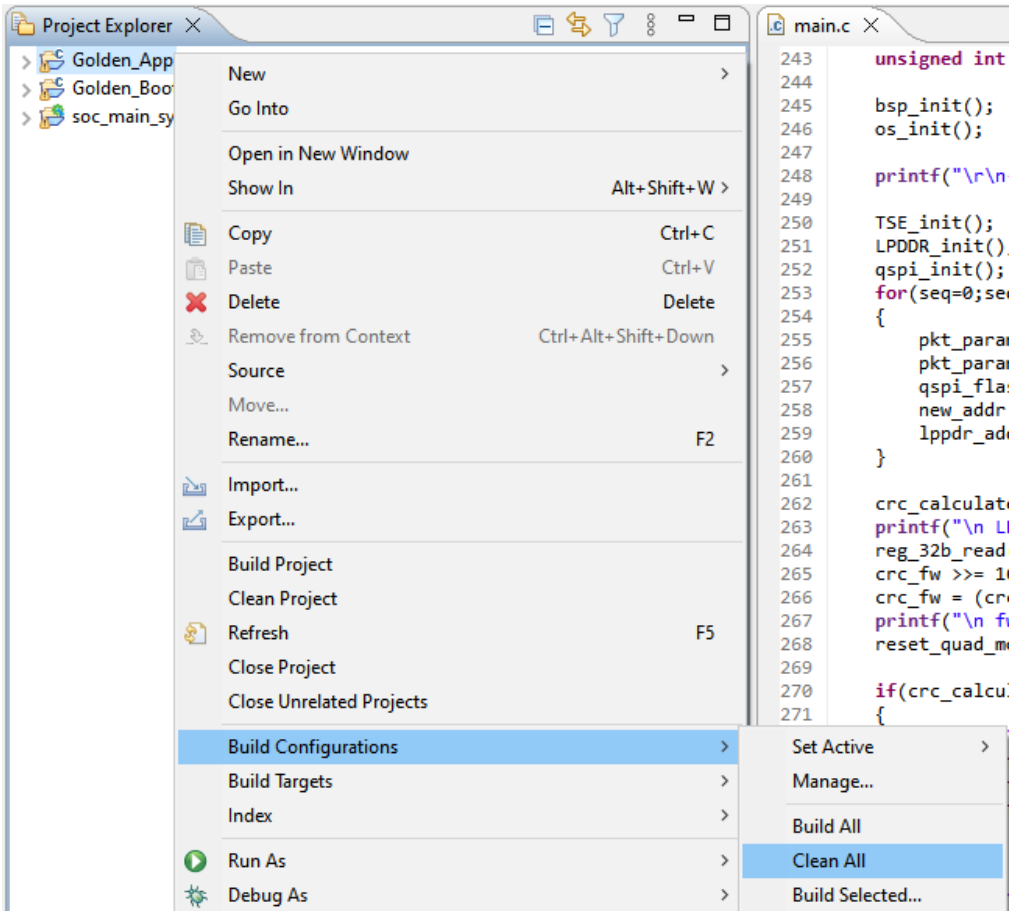

Figure C.35. Clean Project Configurations

9. After selecting the options, observe the console and wait for the process to complete to 100%. After completion, the message shown in Figure C.36 on the console.

Figure C.36. Console

10. After cleaning, right-click on the **Golden_App** and select the option as shown in Figure C.37 to build the project.

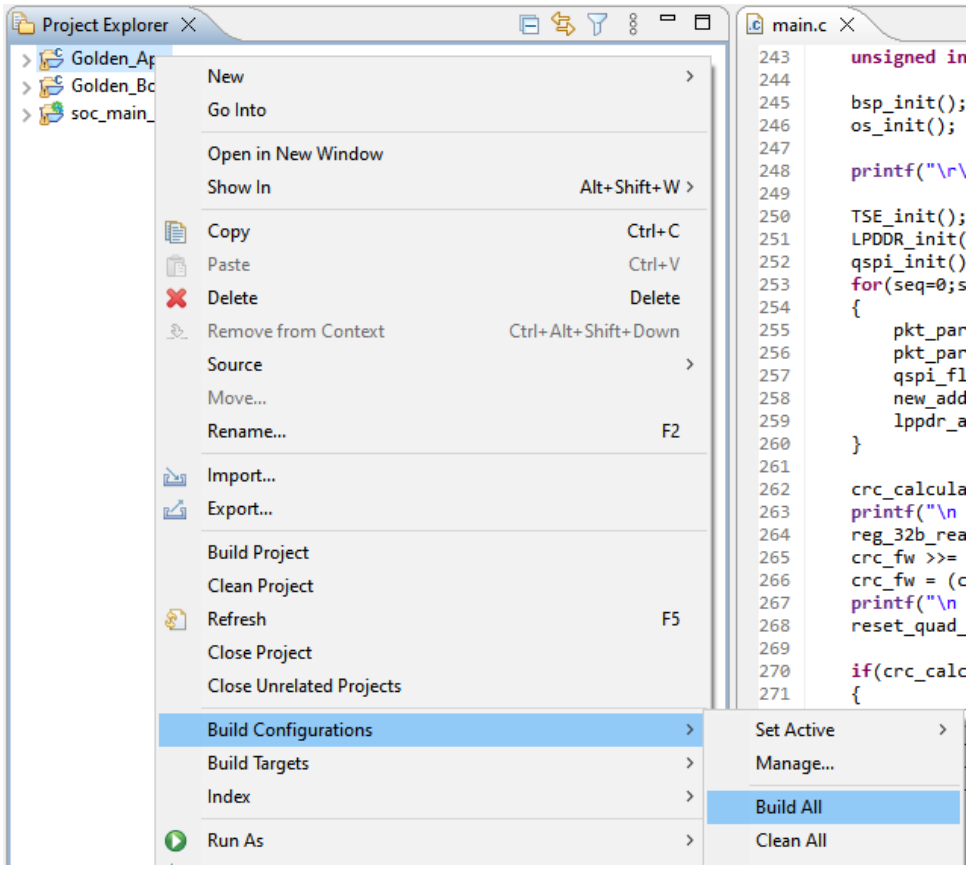

Figure C.37. Build Project

11. Wait for the process to complete to 100%. After completion, the message shown in Figure C.38 appears on the console.

Figure C.38. Completing Process

12. Locate the binary file to below path: **\MainSystem\Golden_MainSystem\Golden_App\Debug**.
13. Right-click on the bootloader project folder **Golden_Bootloader_Cproj** and select the option as shown in Figure C.39 to clean the project before building.

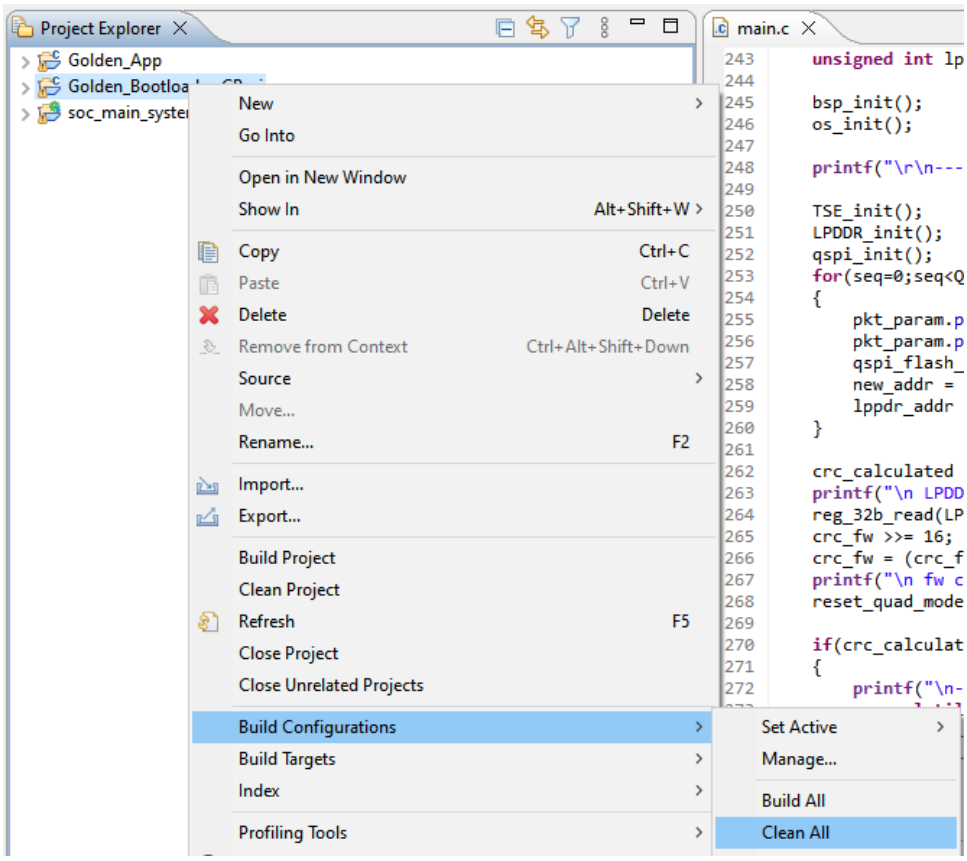

Figure C.39. Clean All Configurations

14. After selecting the option as shown in Figure C.39, observe the console and wait for the process to reach 100%. After completion, the message shown Figure C.40 appears on the console.

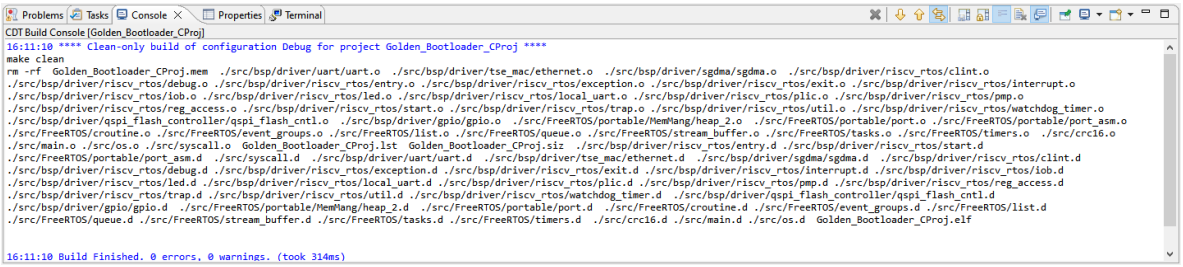

Figure C.40. Console

15. After cleaning, right-click on *Golden_Bootloader_Cproj* and select the option as shown in Figure C.41 to build the project.

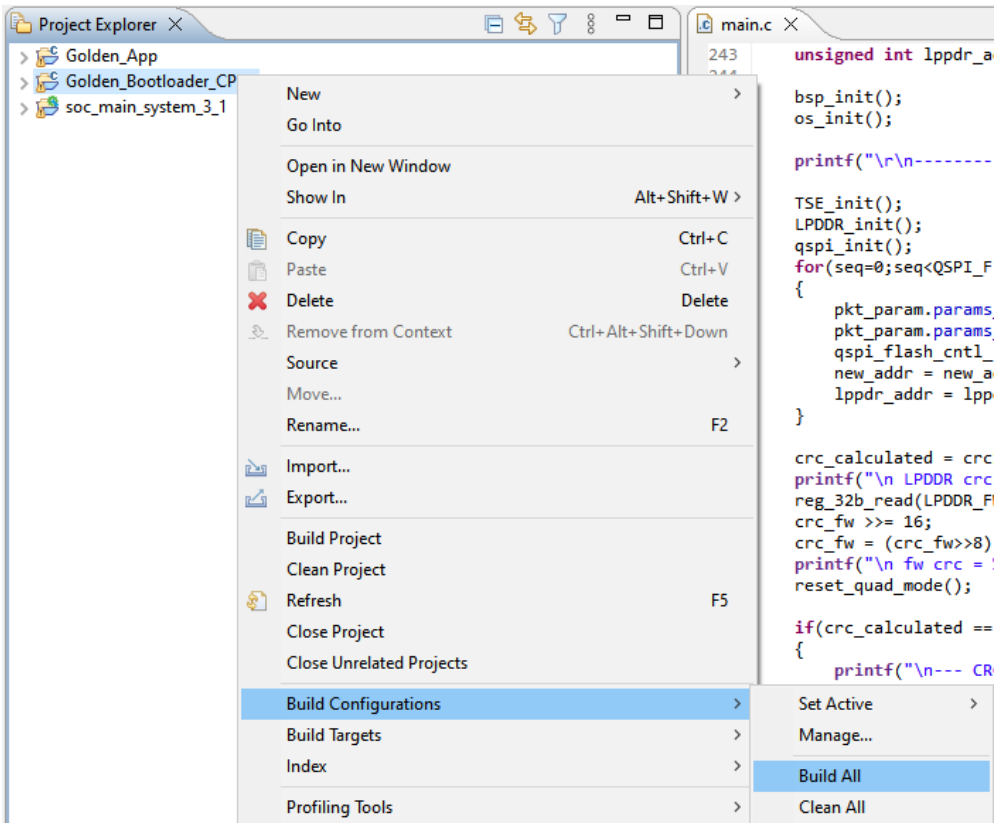

Figure C.41. Build All Configurations

16. Wait for the process to reach 100%. After completion, the message shown in Figure C.42 appears on the console.

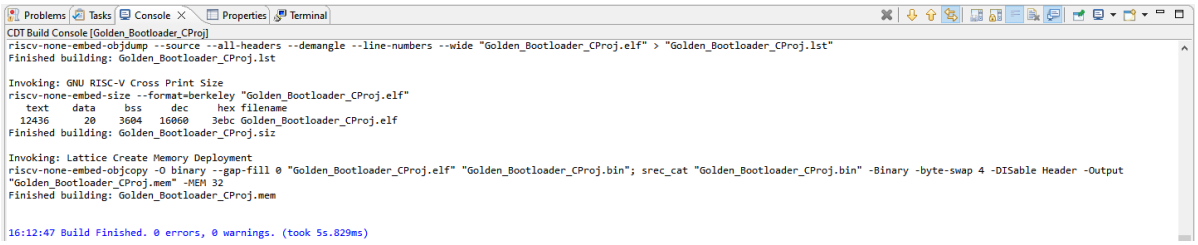

Figure C.42. Completing Process

17. Locate the binary file: **\MainSystem\Golden_App\Golden_Bootloader_CProj\Debug**.

C.3.3. Node System

To generate the binary in the node system, perform the steps below:

1. Double-click **Lattice Propel SDK 2023.2** to open the dialogue box as shown in Figure C.43.

Figure C.43. Propel Application

2. To select the workspace, browse to the template location: **\NodeSystem**. Click the **Launch** button to launch the workspace.

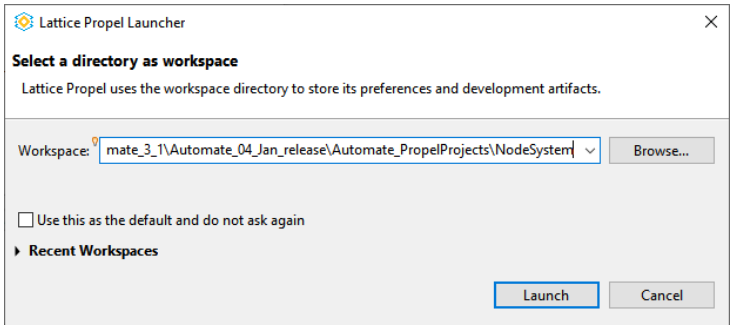

Figure C.44. Select Directory

3. Click **Import projects** or go **Import > File** to import firmware project template.

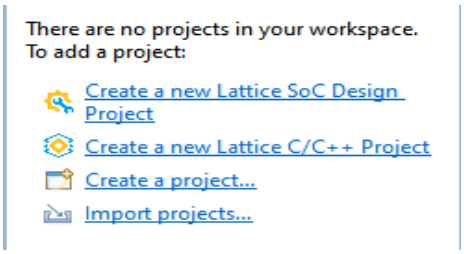

Figure C.45. Import Project

4. Select Existing Project in Workspace from General list and click on next as shown in below fig.

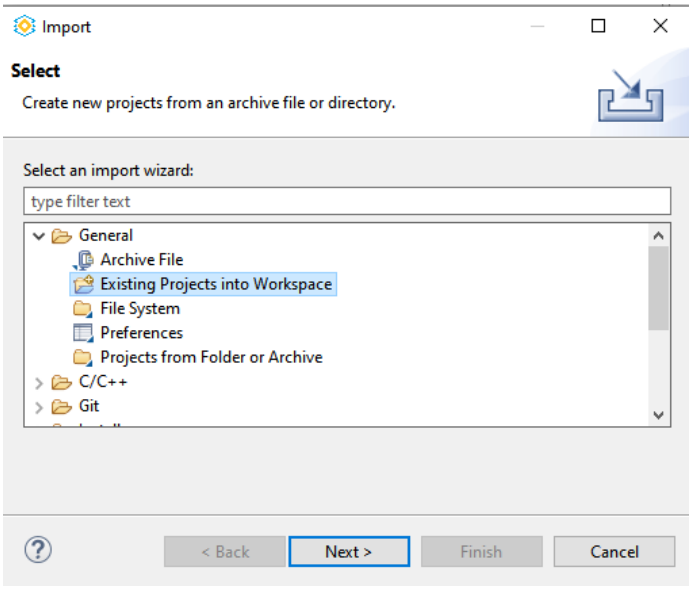

Figure C.46. Existing Project into Workspace

5. Select root directory and browse template location.
6. Select project as shown in below: \NodeSystem
7. Click **Finish**.

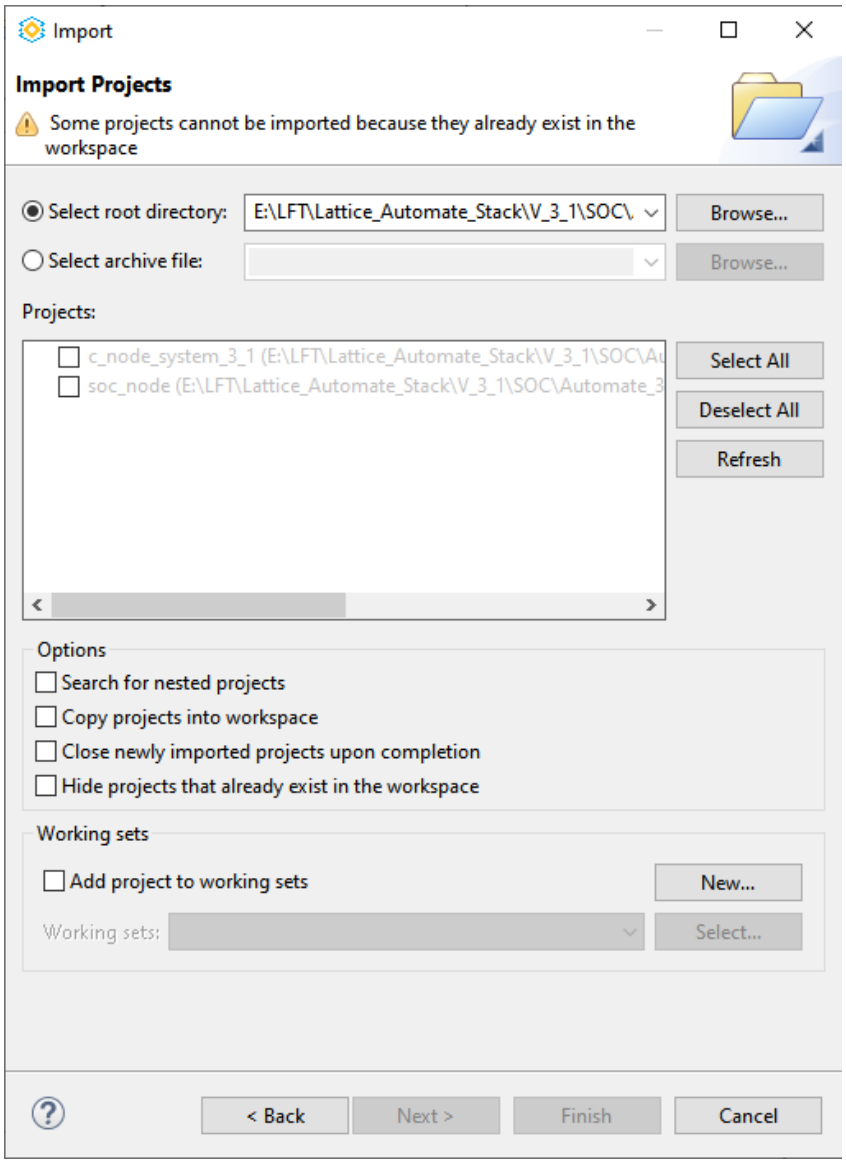

Figure C.47. Select Project

8. Right-click on the firmware project folder “c_node_system_3_1” and select the option as shown in below fig. to clean the project before building.

   Note: If you are doing the fresh patch installation, proceed to Build All configurations directly.

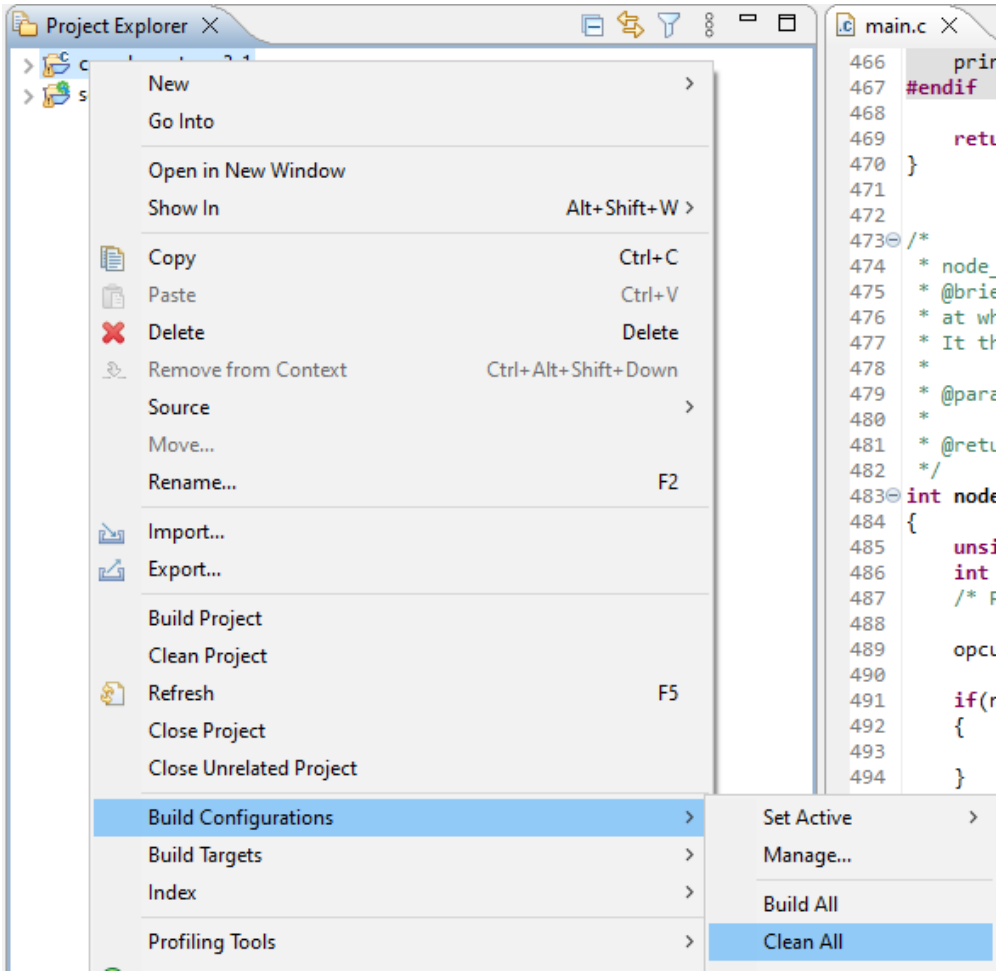

Figure C.48. Clean All

9. After selecting the option as shown in Figure C.48, observe the console and wait for the process to reach 100%. After completion, the message shown in Figure C.49 appears on the console.

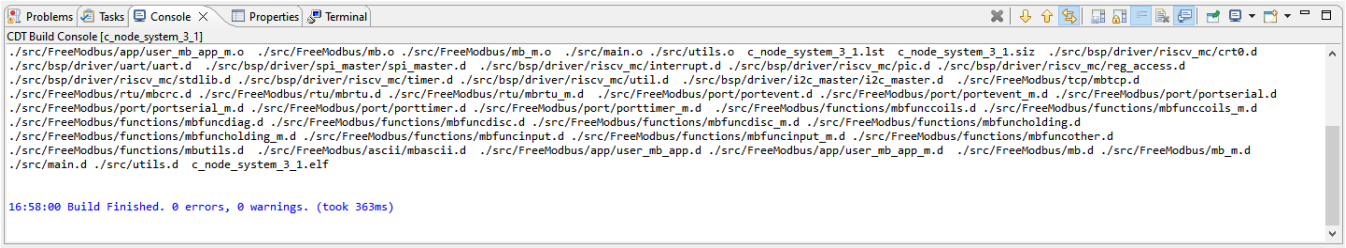

Figure C.49. Console

10. After cleaning, right-click on *c_node_system_3_1* and select the option as shown in Figure C.50 to build the project.

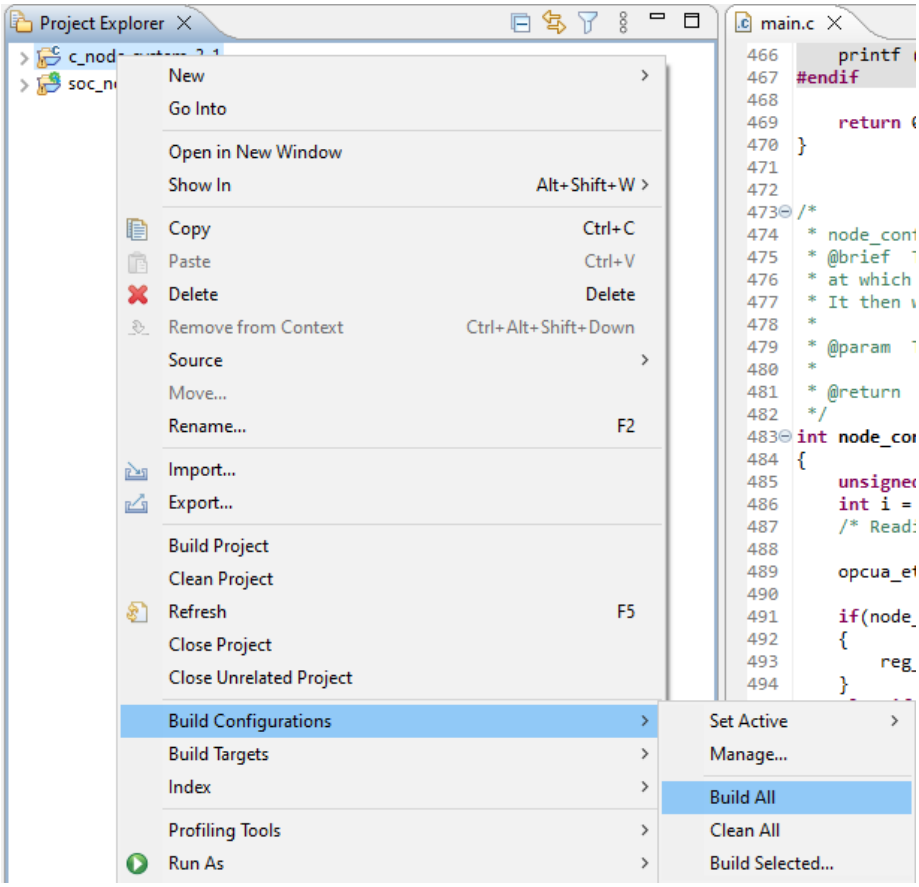

Figure C.50. Build All

11. Wait for the process to reach 100%. After completion, the message shown in Figure C.51 appears on the console.

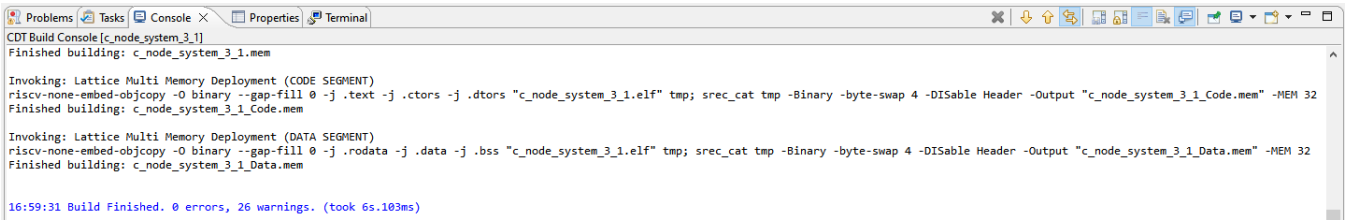

Figure C.51. Completing Process

12. Locate the binary file and .mem file in this path: **\NodeSystem\node_system_3_1\c_node_system_3_1\Debug**.

C.4. Generating the Bit File

C.4.1. Primary Main System

To generate the bit file in the primary main system, perform the steps below:

1. Open the Propel builder 2023.2 tool.
2. Click on the open design symbol and go to this path: Main_System\Primary_MainSystem\soc_main_system_3_1\soc_main_system_3_1. If you do not have the Propel patch, open directly from where project is located. Make sure that there is no space in folder name.
3. Select the **soc_main_system_3_0.sbx** file and the design opens.

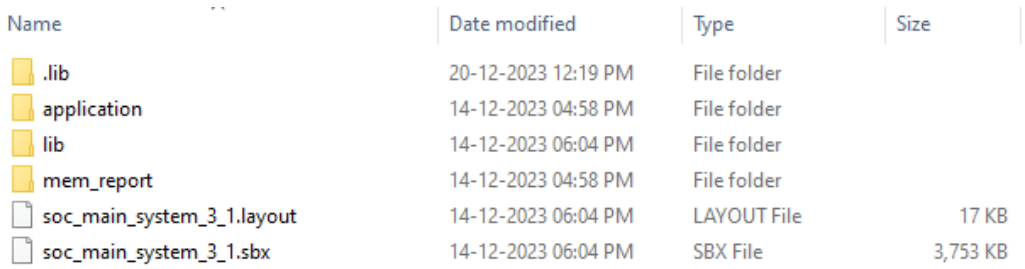

Figure C.52. soc_main_system.sbx

4. Double-click on the **system0_inst**.A pop-up appears on the screen as mentioned below.

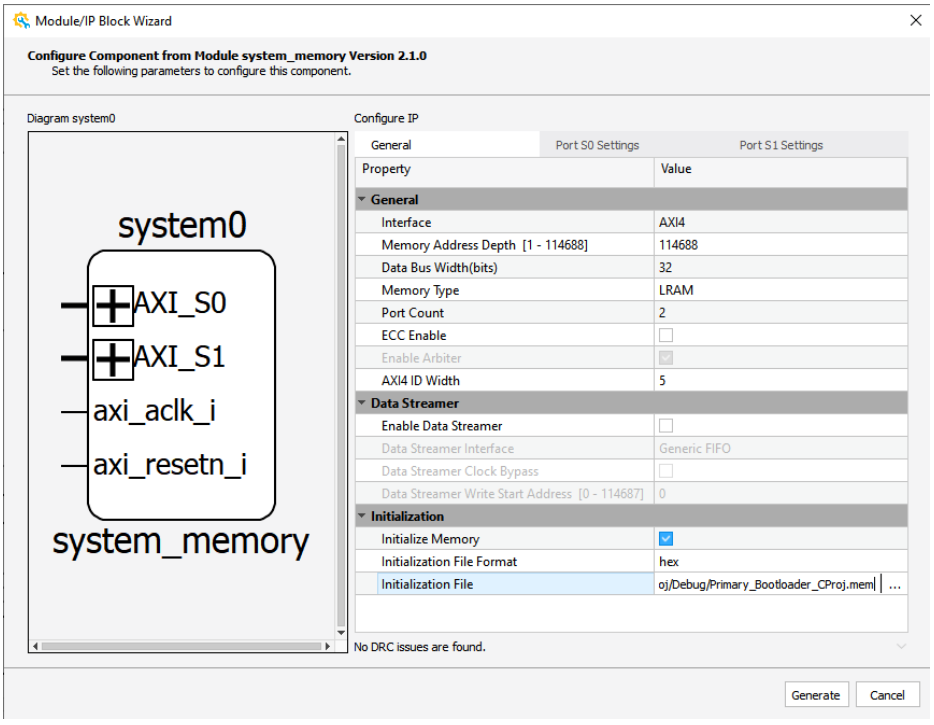

Figure C.53. System Initialization File

5. Initialize Data memory with generated Primary_Bootloader_Cproj.mem file in **\Main_System\Primary_MainSystem\Primary_Bootloader_CProj\Debug** folder of Primary_Bootloader_CProj.
6. Click the **Validate** button.

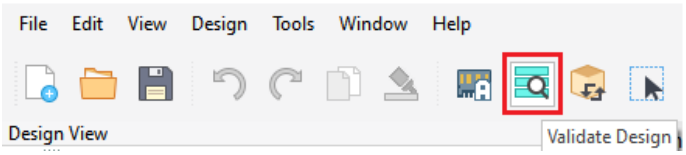

Figure C.54. Validate Button

7. Click the **Generate** button.

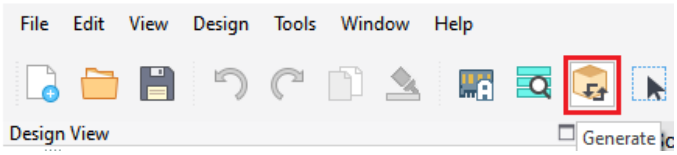

Figure C.55. Generate SGE Button

8. Open the Radiant tool from the Propel Builder interface or open directly.

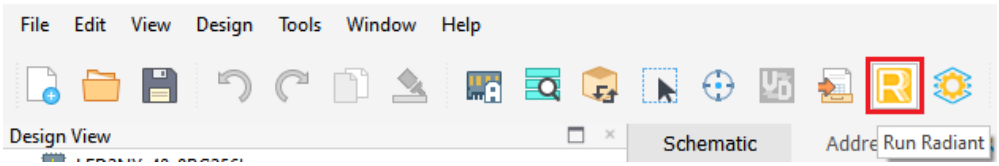

Figure C.56. Radiant Tool Button

Note: To open the Radiant project directly, perform the following:

a. Go to the folder and open the *.rdf file: \Main_System\Primary_MainSystem\soc_main_system_3_1.

b. Select the soc_main_system_3_1.rdf file and the project opens.

Figure C.57. soc_main_system.rdf file

c. Double-click LFCPNX-100-9LFG672I.

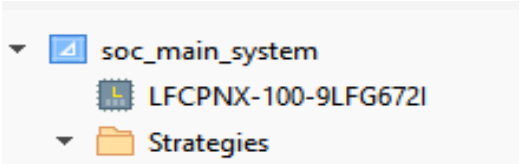

Figure C.58. LFCPNX-100-9LFG672I

9. Apply the following settings:
   - Family: **LFCPNX**
   - Device: **LFCPNX-100**
   - Operating Condition: **Industrial**
   - Package: **LFG672**
   - Performance Grade: **9_High-Performance_1.0V**
   - Part Number: **LFCPNX-100-9LFG672I**

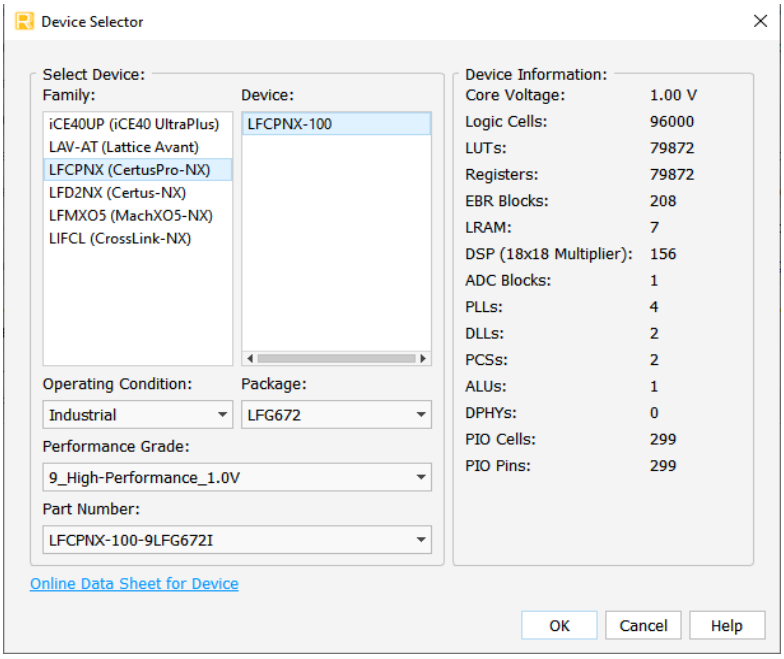

Figure C.59. Lattice Radiant Device Selector for Main System

10. Change value of Frequency to **200 MHz** as shown in Figure C.60.

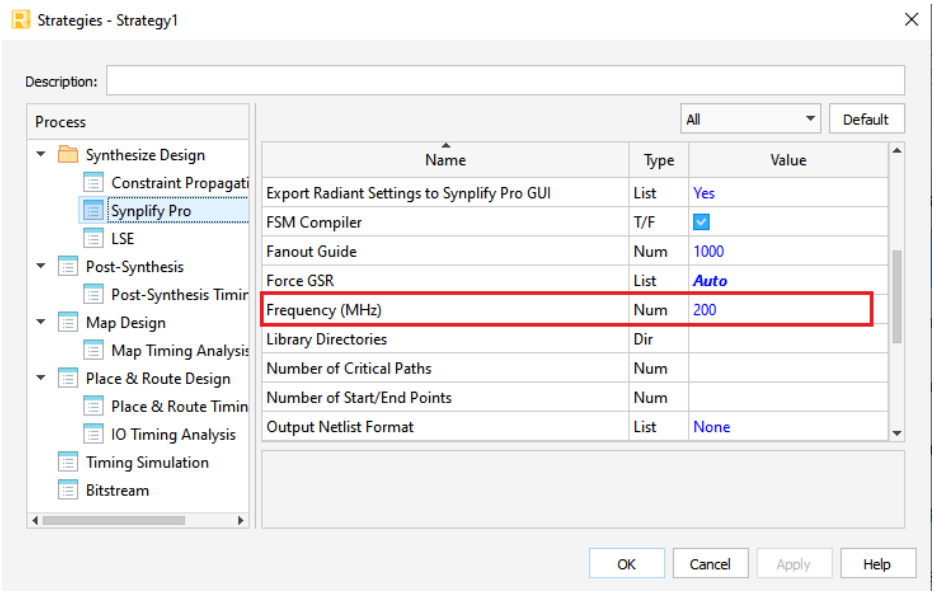

Figure C.60. Strategy for Build Generation for Main System

11. Click the Strategy and go to **Map Design > Map Timing Analysis**. Select the highlighted settings as shown in Figure C.61.

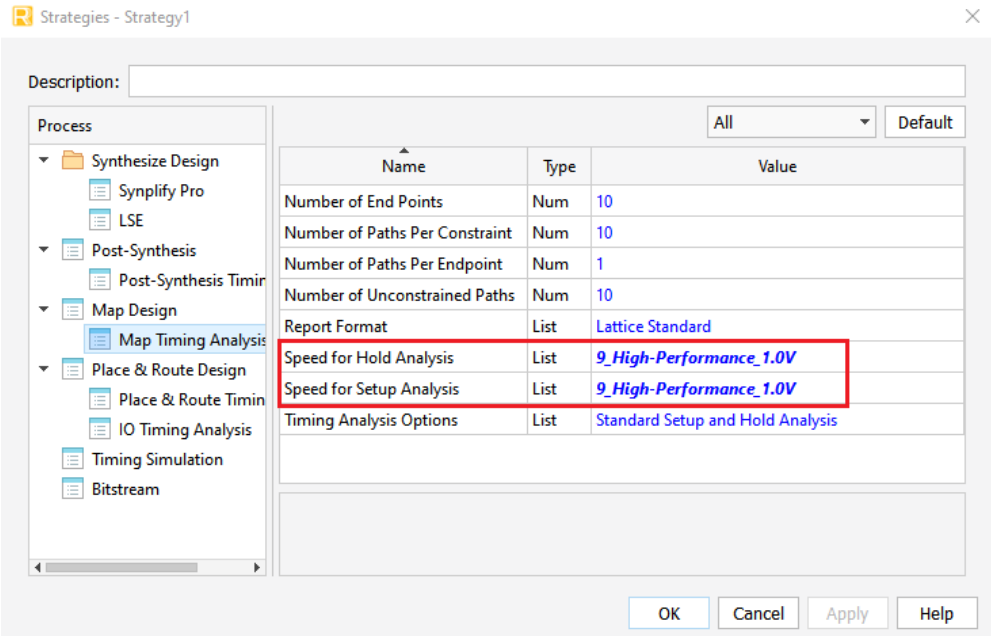

Figure C.61. MAP Analysis Setting for Main System Bit File Generation

12. Select **Place and Route Design** and only apply the settings shown in Figure C.62.

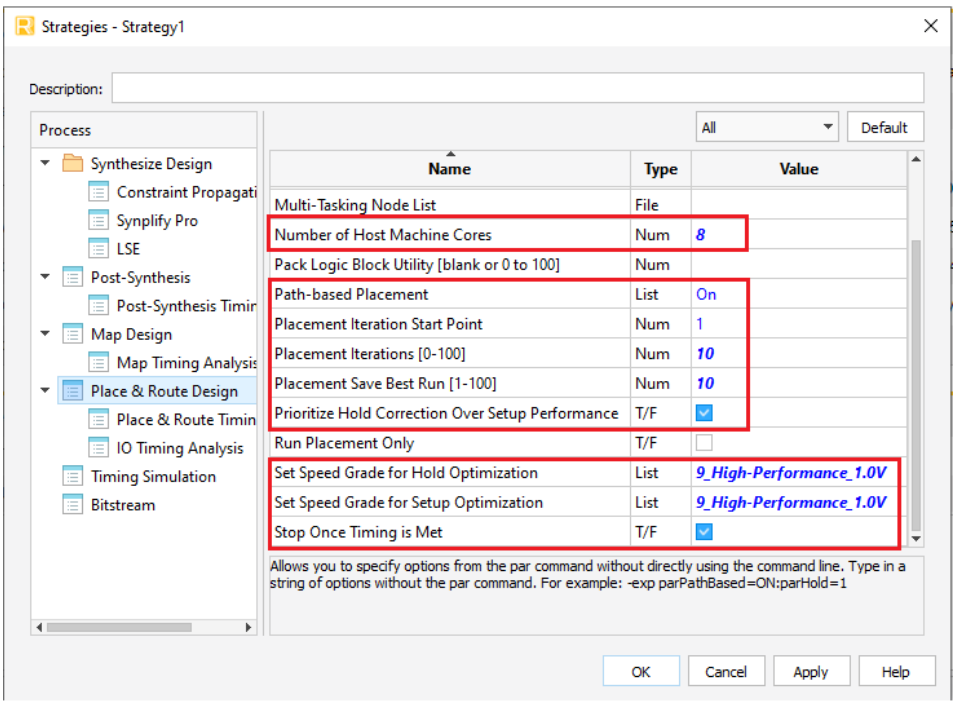

Figure C.62. PAR Setting for Main System Bit File Generation

13. Select **Place and Route Timing Analysis** and only apply the settings shown in Figure C.63.

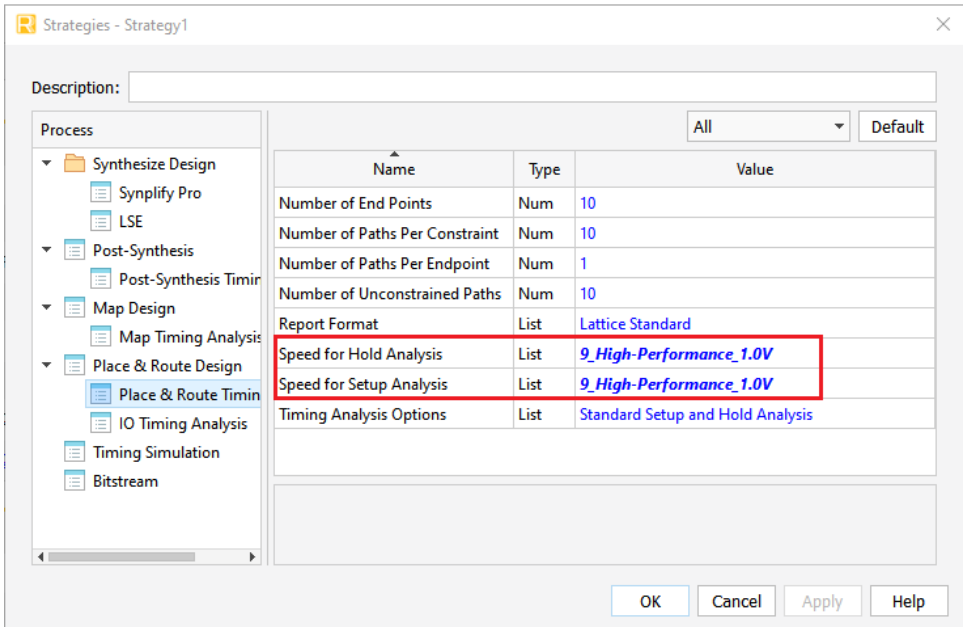

Figure C.63. PAR Timing Analysis Setting for Main System Bitfile Generation

14. Go to **Bitstream** and select the **IP Evaluation** checkbox if you want to generate a non-licensed bit file. Do not check the box if you want to generate a licensed bit file.

    Note: You must request for the license file from the Lattice Semiconductor website.

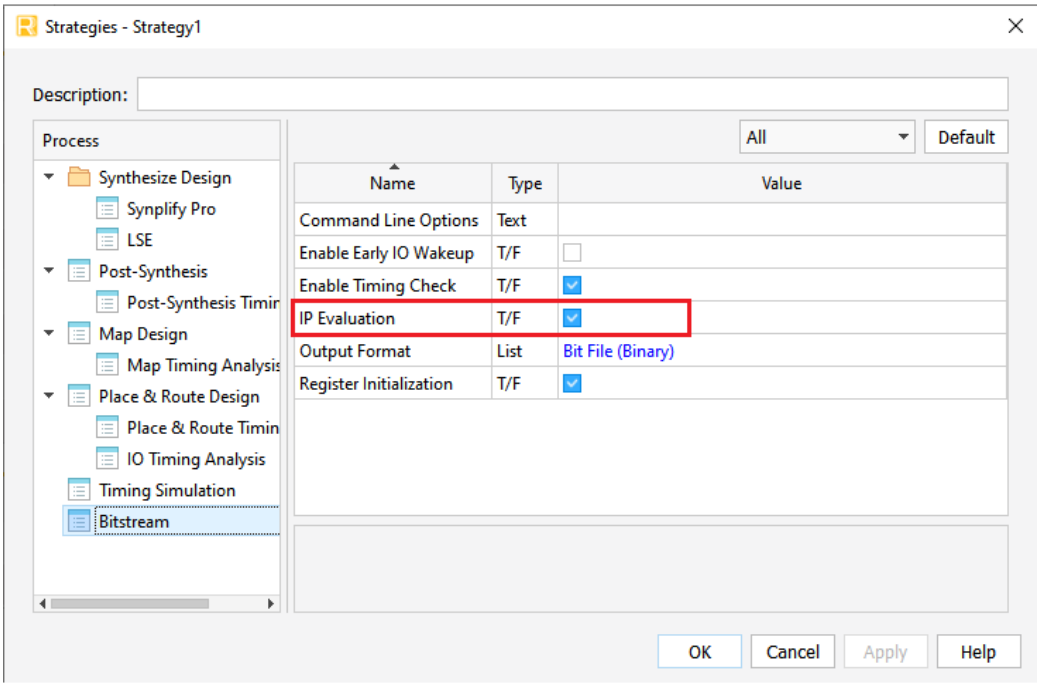

Figure C.64. IP Evaluation

15. Click **Run All** to generate the bit file. Wait for the bit generation and check the output logs.

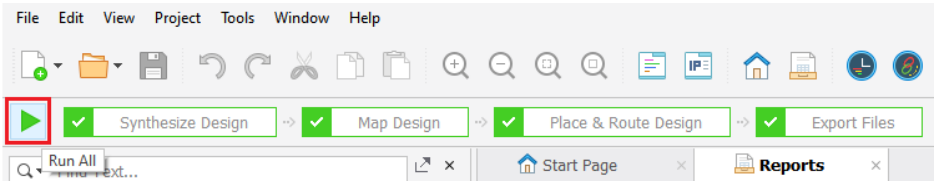

Figure C.65. Run All Button

16. Locate the bitstream file: **\Main_System\Primary_MainSystem\soc_main_system_3_1\impl_1**.

Figure C.66. Bitstream File

C.4.2. Golden Main System

To generate the bit file in the golden main system, perform the steps below:

1. Open the Propel Builder 2023.2 tool.
2. Click the open design symbol and go to this path: **\Main_System\Golden_MainSystem\soc_main_system_3_1\soc_main_system_3_1**. If do not have the Propel patch, open directly from where the project is located. Make sure that there is no space in the folder name.
3. Select the **soc_main_system_3_0.sbx** file and the design window opens.

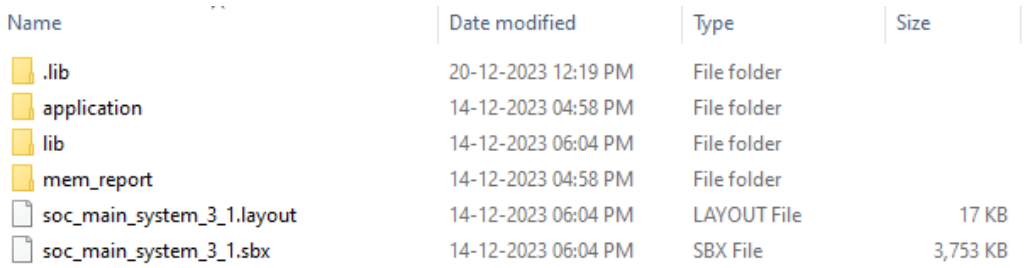

Figure C.67. soc_main_system.sbx

4. Double-click on the **system0_inst.** A pop-up appears on the screen as mentioned below.

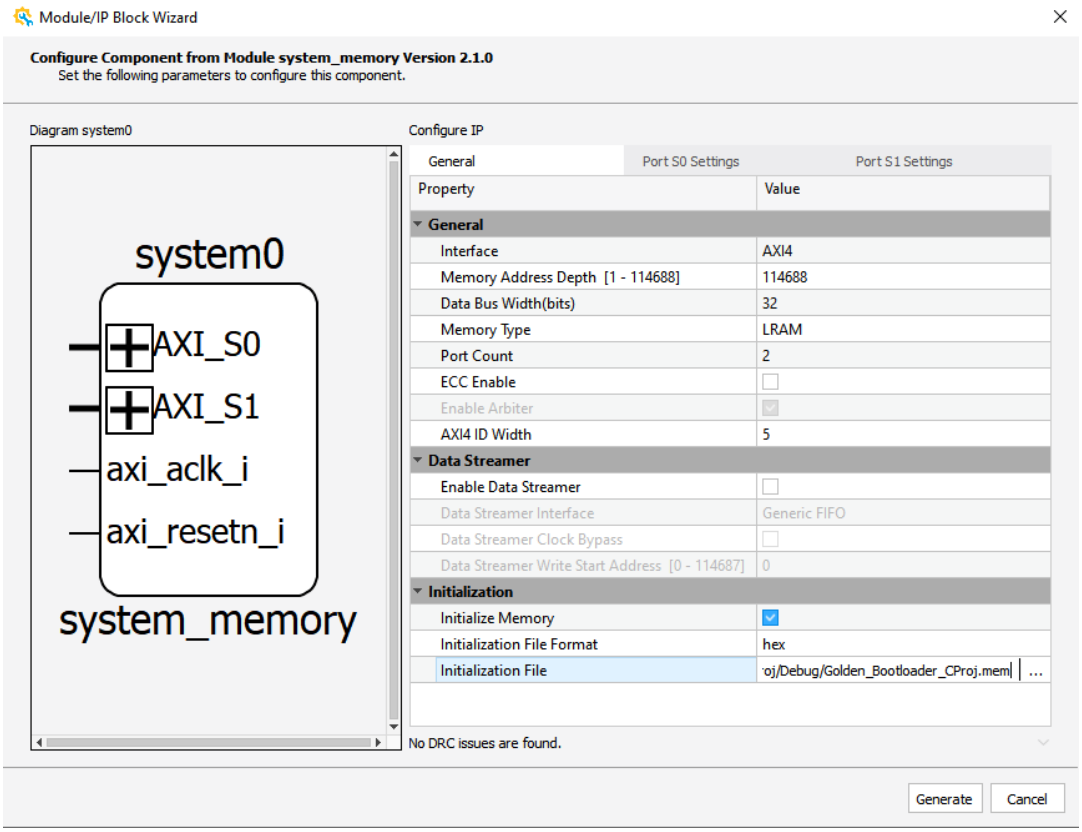

Figure C.68. System Initialization File

5. Initialize Data memory with generated Golden_Bootloader_Cproj.mem file in **\Main_System\Golden_MainSystem\Golden_Bootloader_CProj\Debug** folder of Golden_Bootloader_CProj.
6. Click the **Validate** button.

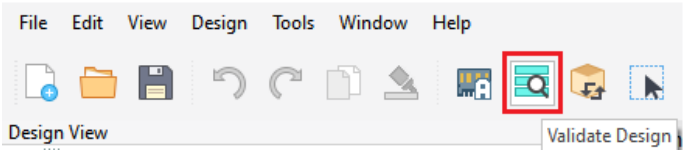

Figure C.69. Validate Button

7. Click the **Generate** button.

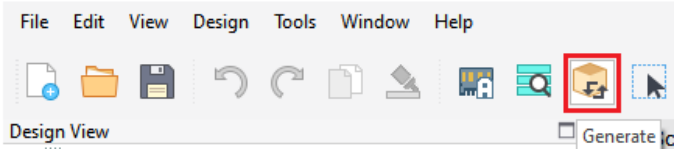

Figure C.70. Generate SGE Button

8. Open the Radiant tool from the Propel Builder interface or open directly.

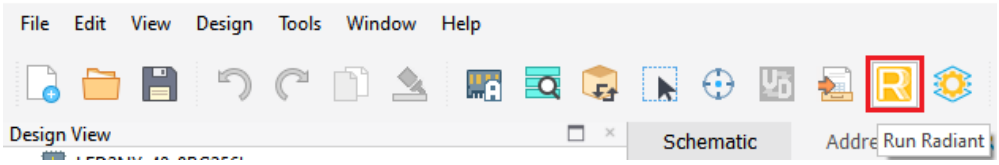

Figure C.71. Radiant Tool Button

Note: To open the Radiant project directly, perform the following steps:

a. Go to the folder and open the *.rdf file: **Main_System\Golden_MainSystem\soc_main_system_3_1**.

b. Select the soc_main_system_3_1.rdf file and the project opens.

Figure C.72. soc_main_sysyem.rdf File

c. Double-click **LFCPNX-100-9LFG672I**.

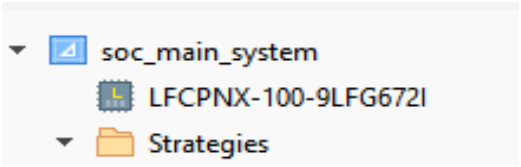

Figure C.73. LFCPNX-100-9LFG672I

9. Apply the following settings:
   - Family: **LFCPNX**
   - Device: **LFCPNX-100**
   - Operating Condition: **Industrial**
   - Package: **LFG672**
   - Performance Grade: **9_High-Performance_1.0V**
   - Part Number: **LFCPNX-100-9LFG672I**

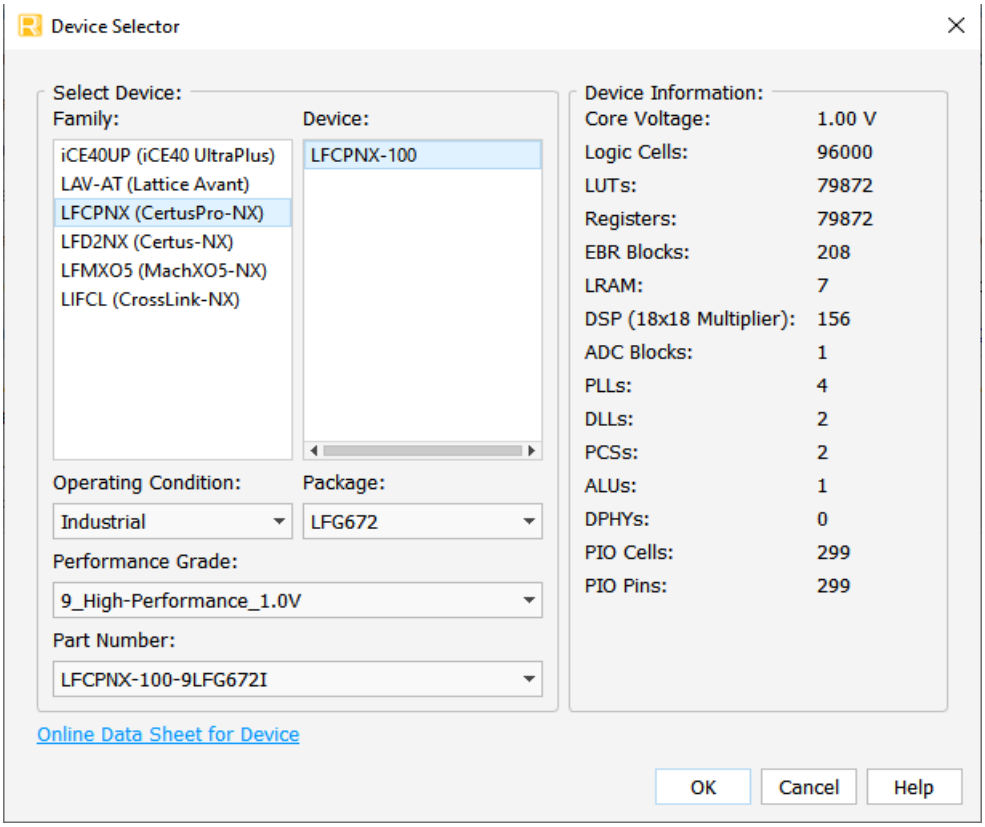

Figure C.74. Lattice Radiant Device Selector for Main System

10. Change value of Frequency to **200 MHz** as shown in Figure C.75.

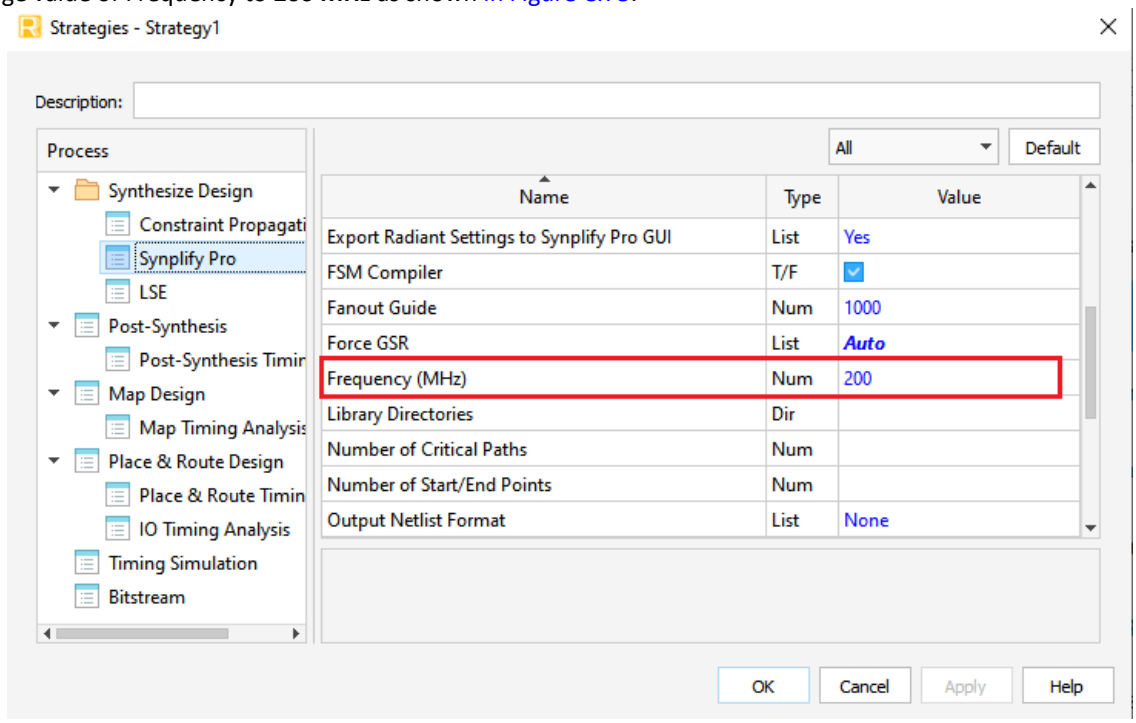

Figure C.75. Strategy for Build Generation for Main System

11. Click the Strategy and go to **Map Design > Map Timing Analysis**. Select the highlighted settings as shown in Figure C.76.

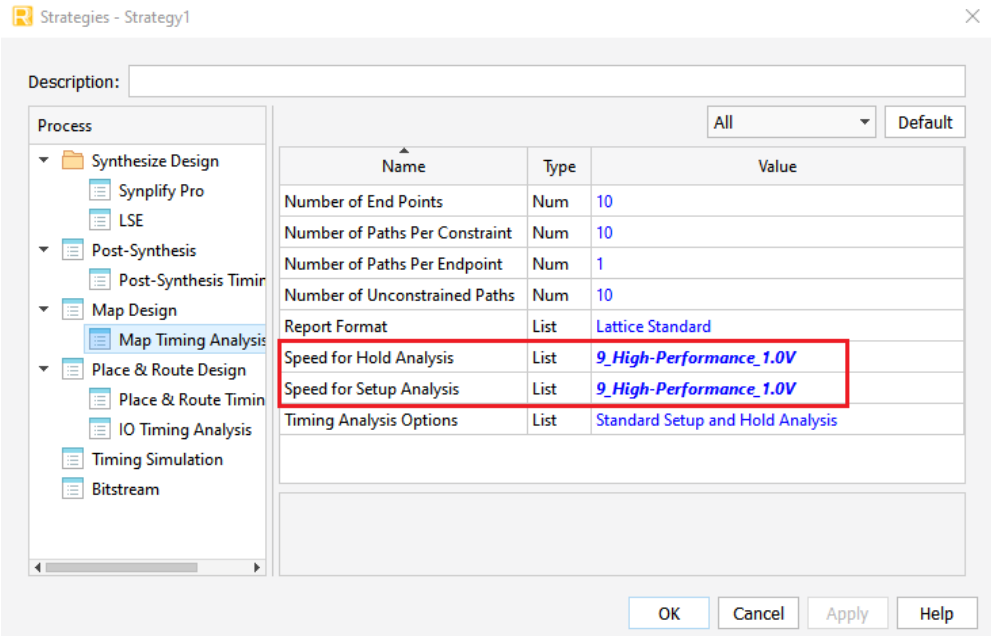

Figure C.76. MAP Analysis Setting for Main System Bit File Generation

12. Select **Place and Route Design** and only apply the settings shown in Figure C.77.

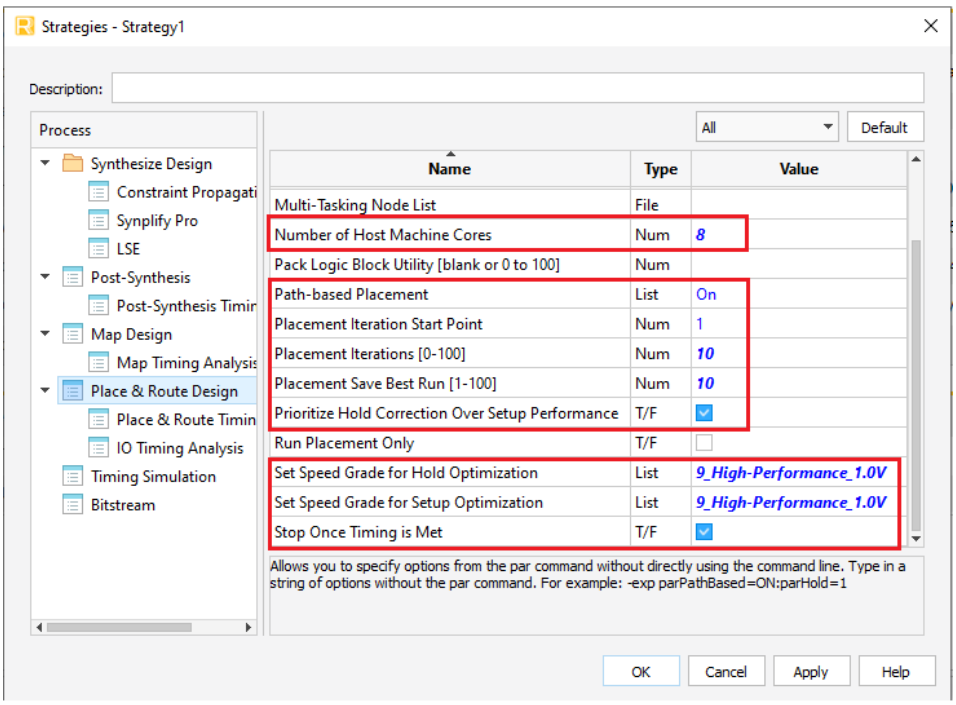

Figure C.77. PAR Setting for Main System Bit File Generation

13. Select **Place and Route Timing Analysis** and only apply the settings shown in Figure C.78.

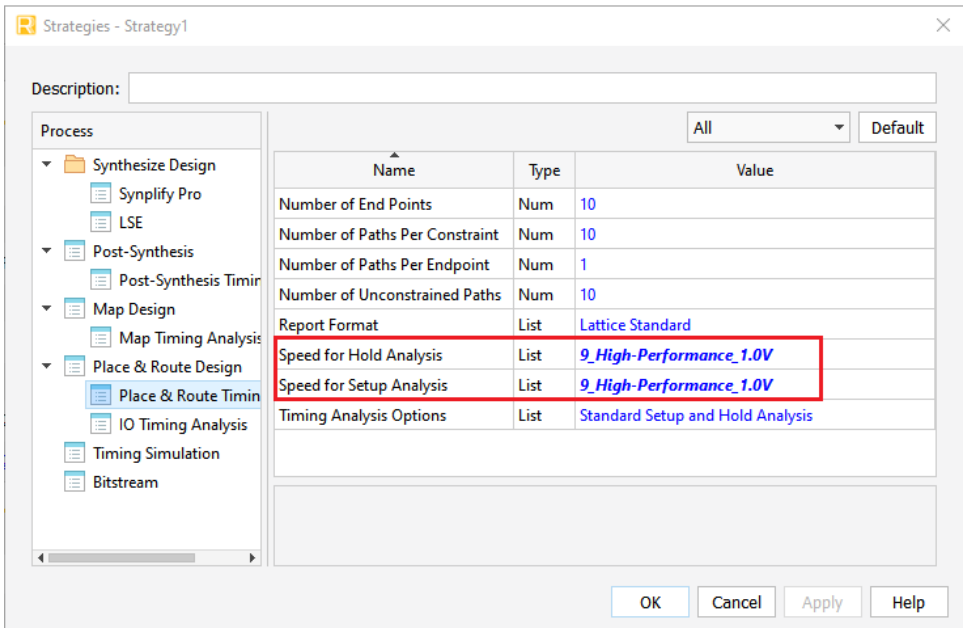

Figure C.78. PAR Timing Analysis Setting for Main System Bitfile Generation

14. Go to **Bitstream** and select the **IP Evaluation** checkbox if you want to generate a non-licensed bit file. Do not check the box if you want to generate a licensed bit file.
    Note: You must request for the license file from the Lattice Semiconductor website.

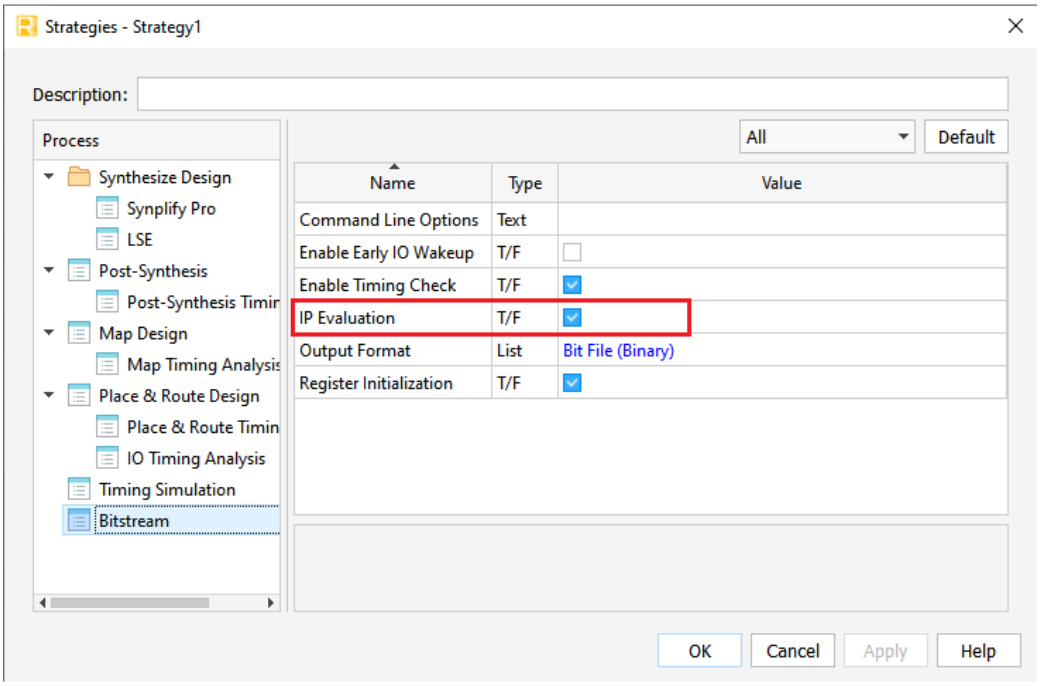

Figure C.79. IP Evaluation

15. Click **Run All** to generate the bit file. Wait for the bit generation and check the output logs.

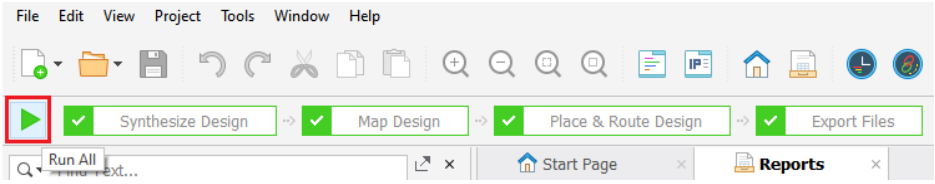

Figure C.80. Run All

16. Locate the bit stream file follow the below path: **\Main_System\Golden_MainSystem\soc_main_system_3_1\impl_1**.

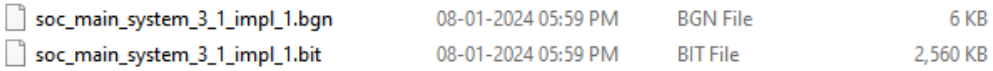

Figure C.81. Bitstream File

C.4.3. Node System

To generate the bit file in the node system, perform the steps below:

1. Open the Propel Builder 2023.2 tool.
2. Click the open design symbol and go to this path: **NodeSystem\node_system_3_1\soc_node\soc_node**.

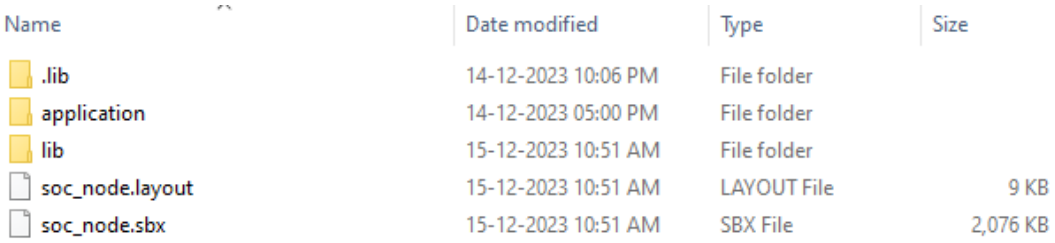

Figure C.82. soc_node.sbx

3. Double-click on **system0_inst**. A pop-up appears on the screen as mentioned below.

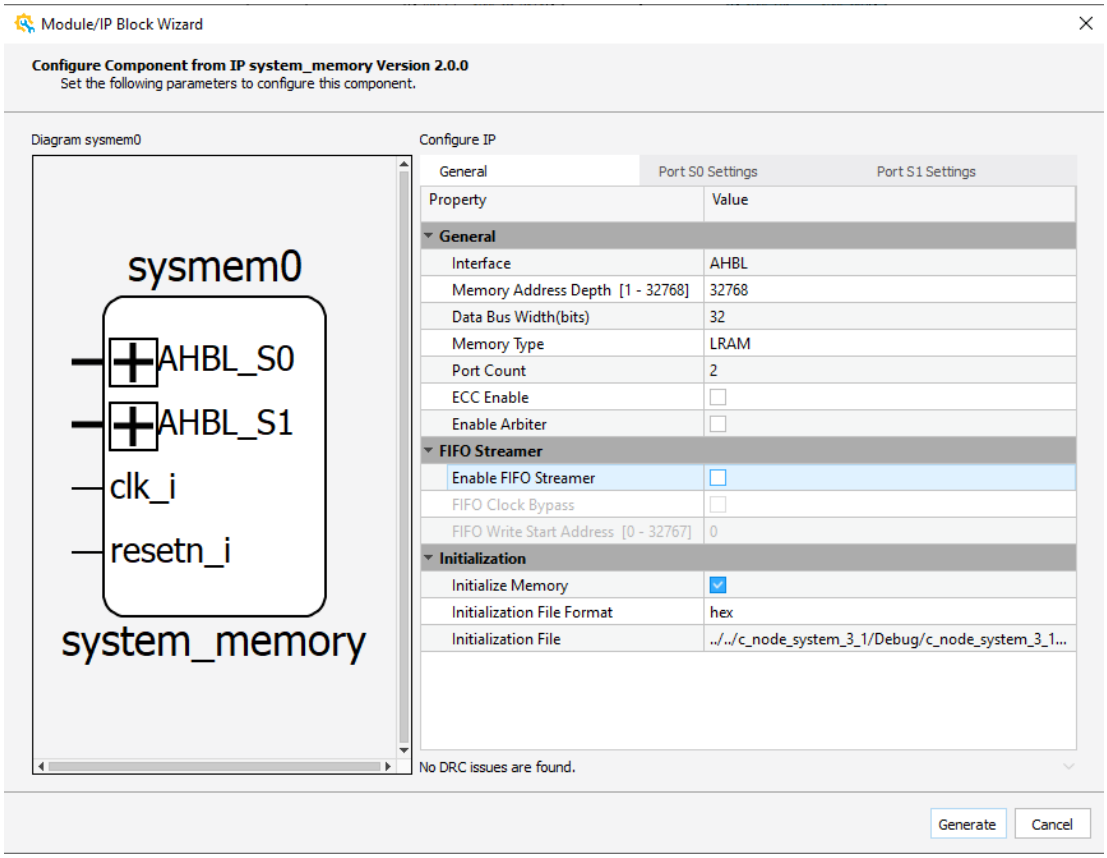

Figure C.83. System0 Initialization

4. Initialize data memory with the generated **c_node_system_Data.mem** file in the debug folder of the C project.
5. Click the **Validate** button.

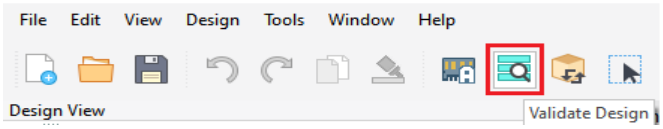

Figure C.84. Validate Button

6. Click the **Generate SGE** button.

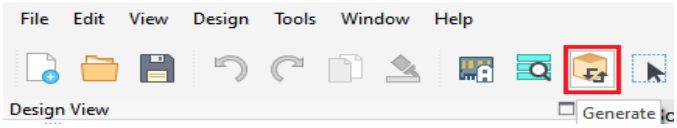

Figure C.85. Generate SGE Button

7. Open the Radiant tool from the Propel Builder interface or open directly.

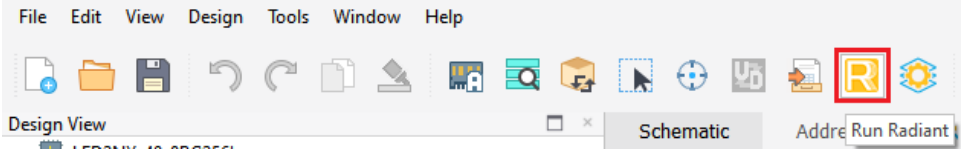

Figure C.86. Radiant Tool Button

Note: To open the Radiant project directly, perform the following:

a. Open the generated Radiant project (**NodeSystem\node_system_3_1\soc_node**) in the Radiant tool.

b. Select the **soc_node.rdf** file and the project opens.

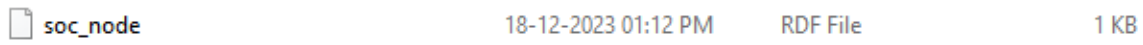

Figure C.87. soc_node.rdf file

c. Click on **LFD2NX-40-8BG256C**.

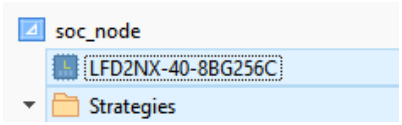

Figure C.88. LFD2NX-40-8BG256C

8. Apply the settings below:

a. Family: **LFD2NX**

b. Device: **LFD2NX-40**

c. Operating Condition: **Commercial**

d. Package: **CABGA256**

e. Performance Grade: **8_High-Performance_1.0V**

f. Part Number: **LFD2NX-40-8BG256C**

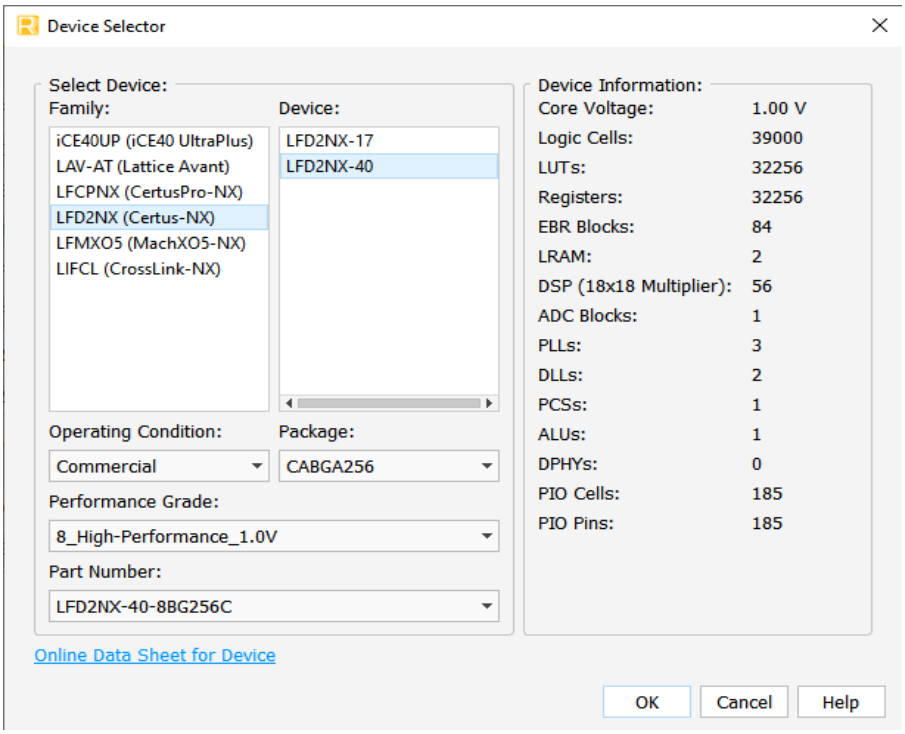

Figure C.89. Lattice Radiant Device Selector for Node System

9. Change value of Frequency to **150 MHz** as shown in Figure C.90.

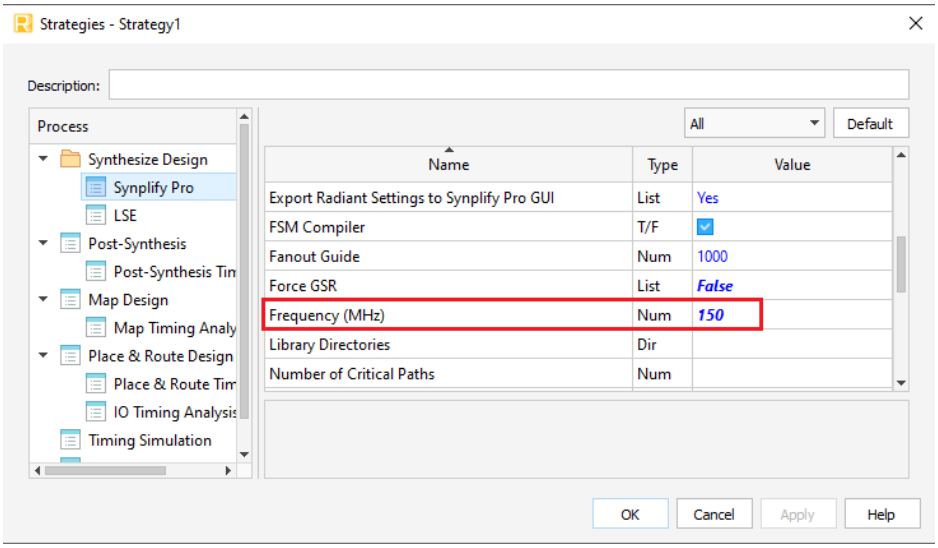

Figure C.90. Strategy for Build Generation for Node System

10. Click the Strategy and go to **Map Design > Map Timing Analysis**. Select the highlighted settings as shown in Figure C.91.

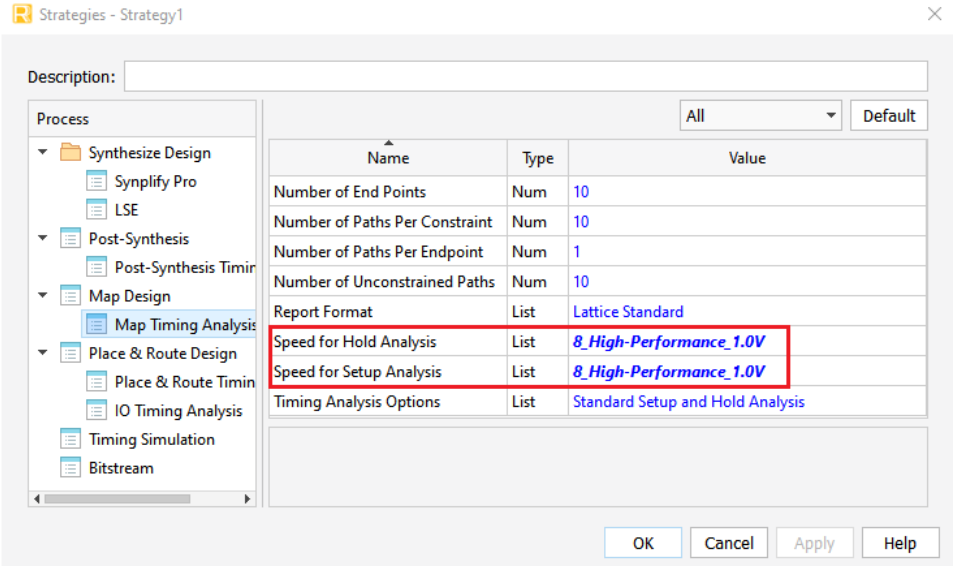

Figure C.91. MAP Analysis Setting for Node System Bit File Generation

11. Select **Place and Route Design** and only apply the settings shown in Figure C.92.

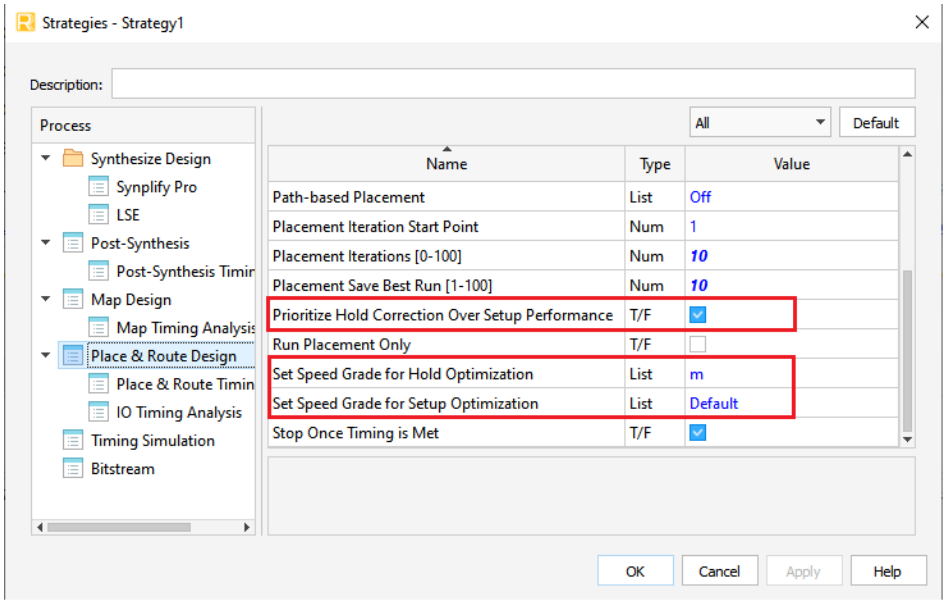

Figure C.92. PAR Setting for Node system Bit File Generation

12. Select **Place and Route Timing Analysis** and only apply the settings shown in Figure C.93.

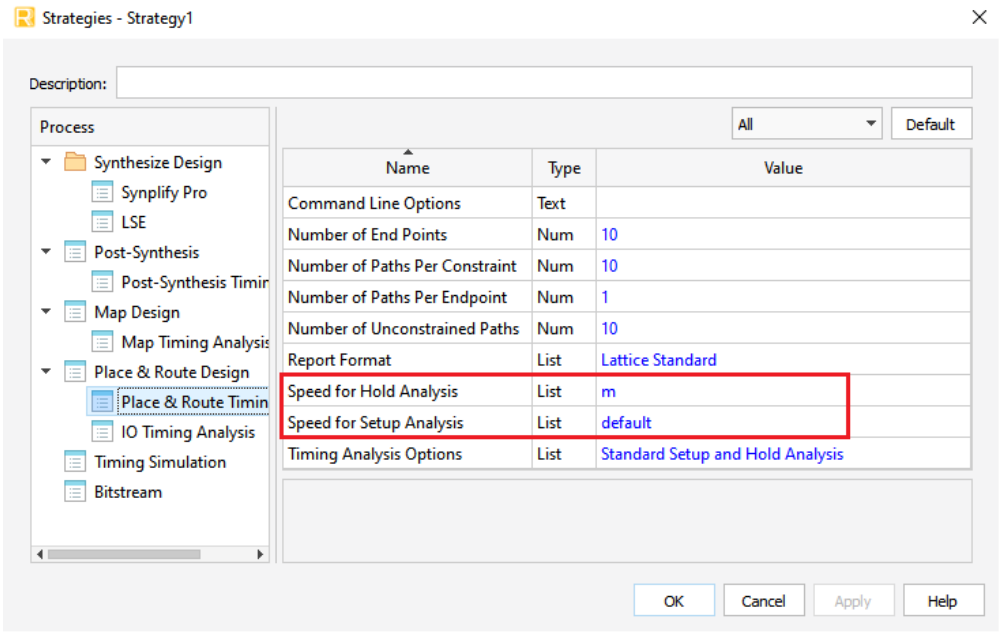

Figure C.93. PAR Timing Analysis Setting for Node System Bit File Generation

13. Go to **Bitstream** and select the **IP Evaluation** checkbox if you want to generate a non-licensed bit file. Do not check the box if you want to generate a licensed bit file.

    Note: You must request for the license file from the Lattice Semiconductor website.

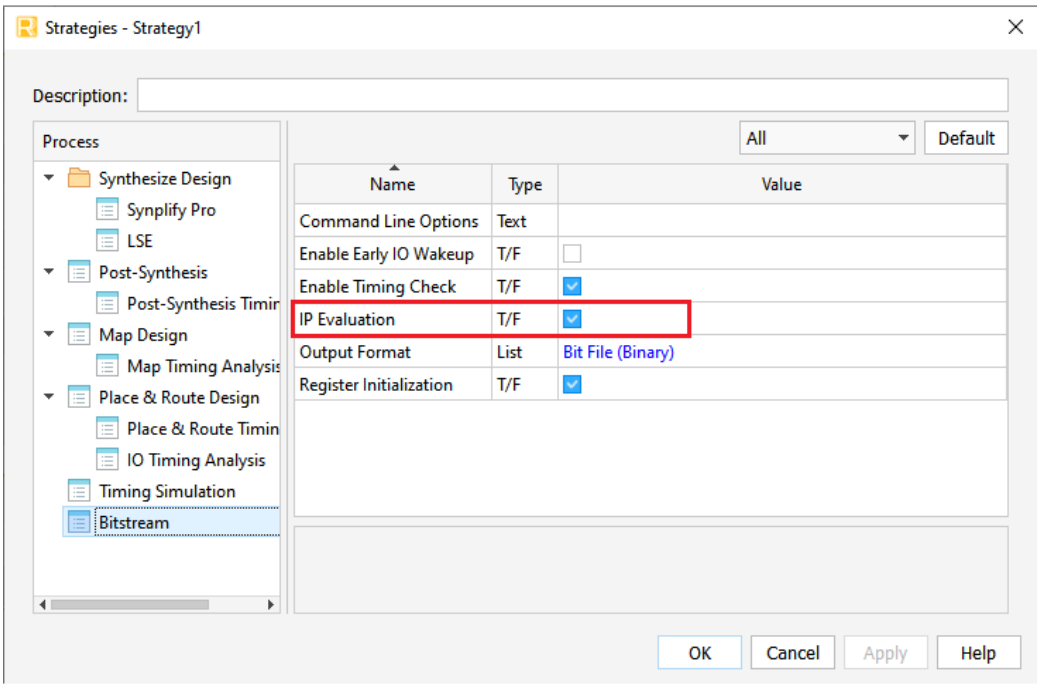

Figure C.94. IP Evaluation

14. Click **Run All** to generate the bit file. Wait for the bit generation and check the output logs.

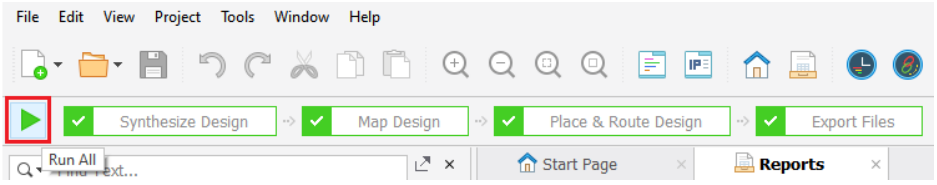

Figure C.95. Run All Button

15. Locate the bit stream file in this path: \NodeSystem\node_system_3_1\soc_node\impl_1.

Figure C.96. Bitstream File

Appendix D. Creating the MCS File

The following provides the steps for generating a Multi-Boot PROM hex file using the Radiant Deployment tool. This procedure is an example for three total bitstream, primary pattern, golden pattern, and alternate pattern 1.

1. Open the Lattice Radiant Programmer and go to **Tools > Deployment Tool**.

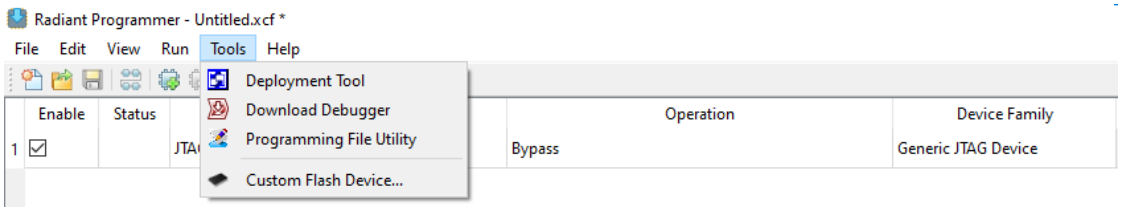

Figure D.1. Deployment tool

2. Select **External Memory** for Function Type and **Advanced SPI Flash** for Output File Type.

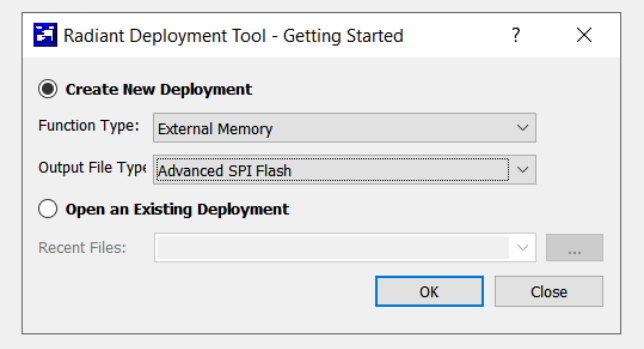

Figure D.2. Creating New Deployment for Multi-Boot

3. Click **OK**.
4. In the Select Input File(s) window, click the **File Name** field to browse and select the primary bitstream file to be used to create the PROM hex file. The device family and device fields auto-populate based on the bitstream files selected. Click **Next**.

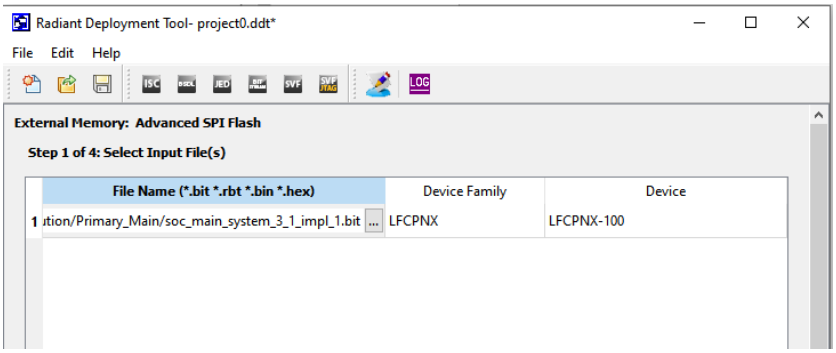

Figure D.3. Select Input File Window

5. In the Advanced SPI Flash Options window, click the **Multiple Boot** tab and select the *Multi-Boot* option. Apply the following settings.
   a. Click the **Golden Pattern** browse button to select the primary pattern bitstream. The starting address of the Golden pattern is automatically assigned. You can change it by clicking on the drop-down menu.
   b. In the number of Alternate Patterns field, select the number of patterns to include through the drop-down menu.
   c. In the Alternate Pattern 1 field, click on the browse button to select the golden pattern bitstream. The starting address of the primary pattern is automatically assigned. You can change it by clicking on drop down menu. The address of next alternate pattern to configure field is automatically populated. This is the pattern that is loaded during the next PROGRAMN/REFRESH event. You can change the pattern by clicking on the drop-down menu.
   d. Click **Next**.

   Note: The starting address of golden pattern should be more than the size of primary pattern and the starting address of alternate pattern 1 must be more than the starting address and size of golden pattern. Otherwise, it shows an error.

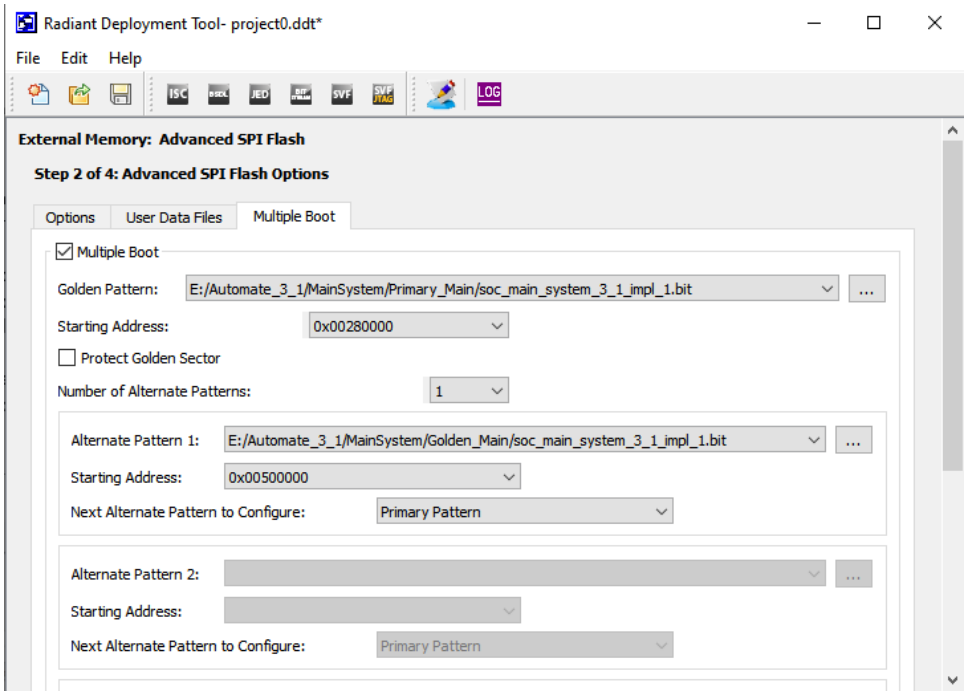

Figure D.4. Advanced SPI Flash Options - Multi-Boot Tab Window

6. In the Select Output File window, specify the name of the output PROM hex file in the **Output File 1** field. Click **Next**.

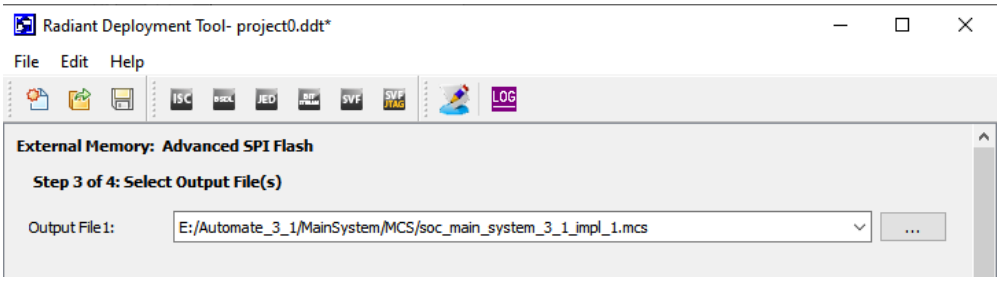

Figure D.5. Select Output File Window

7. Review the summary information in the Generate Deployment window. If everything is correct, click the **Generate** button. The generate deployment pane indicates the PROM file is successfully generated.
8. Go to **File > Save** to save the deployment settings.

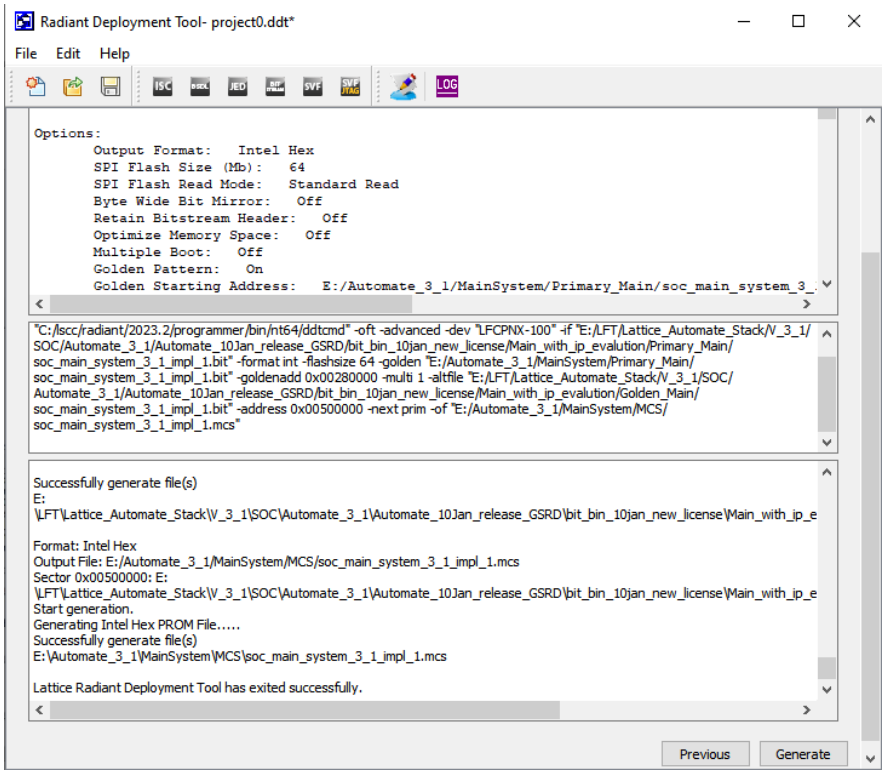

Figure D.6. Generate Deployment Window

9. Once the setting is saved, you can program the .mcs file in the external flash using the Radiant Programmer.

References

- Lattice Automate

Other references:

- Lattice Insights for Lattice Semiconductor training courses and learning plans
- Lattice Radiant FPGA design software

Technical Support Assistance

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

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

Revision History

Revision 1.0, April 2024

Section	Change Summary
All	Production release.

www.latticesemi.com