Automate 4.0

Reference Design

FPGA-RD-02302-1.0

February 2025

Disclaimers

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

Inclusive Language

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

Contents

Figures

Tables

Abbreviations in This Document

A list of abbreviations used in this document.

Abbreviation	Definition
AHBL	Advanced High-performance Bus-Lite
AI	Artificial Intelligence
API	Application Programming Interface
BLDC	Brushless DC
CCU	CNN Co-Processor Unit
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DMA	Direct Memory Access
FIFO	First-In-First-Out
GMII	Gigabit Media Independent Interface
ISR	Interrupt Service Routines
ICMP	Internet Control Message Protocol
LATTE	Lattice Training Environment
ML	Machine Learning
QSPI	Quad Serial Peripheral Interface
RGMII	Reduced Gigabit Media Independent Interface
RISC-V	Reduced Instruction Set Computer-V
RTL	Register-Transfer Level
SGMII	Serial gigabit media-independent interface
UART	Universal Asynchronous Receiver-Transmitter
DDR	Double Data Rate
UDP	User Data gram Protocol
LPDDR4	Low Power Double Data Rate Generation 4
TSEMAC	Tri-Speed Ethernet Media Access Controller

1. Introduction

The Automate 4.0 Reference Design serves as a comprehensive base for developing a wide range of industrial solutions, including industrial Ethernet communication protocols like EtherCAT and Profinet, predictive maintenance, open and closed motor control, and industrial robotics. This reference design is built based on the Lattice Avant™-E GSRD to provide a robust and flexible platform for industrial applications.

The Automate 4.0 reference design is composed of two primary systems: the main system and the node system. Each system plays a crucial role in the overall functionality and performance of the solution stack.

1.1. Automate 4.0 System Architecture Overview

1.1.1. Main System

The main system is built around the Avant-E FPGA. The key components and features of the main system include:

- Avant-E base RISC-V: The base RISC-V based SOC system that handles complex computations and control tasks.
- EtherConnect IP: Developed to facilitate Ethernet communication, enabling seamless data transfer between the main and node systems.
- CNN Co-processor IP: Used for predictive maintenance application, ensuring efficient and accurate performance.
- Interface IP: Provides the necessary interfaces for connecting various peripherals and components within the system.
- Soft RISC-V SoC Platform: A flexible and programmable system-on-chip that supports the integration of custom IP blocks and firmware updates.
- Firmware Updates: Includes LWIP (Lightweight IP) and TCP/IP stack to establish a reliable connection with the host system, allowing for control through the Automate 4.0 user interface.

The main system firmware and IP blocks are designed to enable robust communication and control capabilities, making it a central hub for managing industrial automation tasks. The current reference design utilizes about 18% of FPGA resources leaving almost more than half of the resources for customer specific glue logic and IP.

1.1.2. Node System

The node system utilizes the Certus™-NX FPGA, which is also built on a soft RISC-V SoC platform. Key components and features of the node system include:

- Certus-NX FPGA: Provides a low power FPGA solution for local control and data acquisition tasks.
- Motor Control IP: Implements advanced algorithms for controlling motor speed, direction, and torque, ensuring precise and efficient motor operation.
- Encoder IP: Used to monitor and provide feedback on motor position, enabling closed-loop control for enhanced accuracy and performance.
- EtherConnect IP: Facilitates Ethernet communication with the main system, ensuring synchronized operation and data exchange.

The node system is designed to operate in conjunction with the main system, providing localized control and feedback for motor operations and other industrial processes.

1.2. Advantages of the Automate 4.0 Reference Design

- Ease of Use: The reference design is user-friendly, with all necessary components and connections pre-configured. This enables customers to quickly bring their systems online, often within a few hours. The modular nature of the design allows for easy customization and scalability, making it suitable for a wide range of industrial applications.
- Cost Savings: By providing a ready-to-use reference design, the Automate 4.0 reference design significantly reduces development time and costs. You can leverage the pre-developed IP and firmware updates to accelerate their project timelines and reduce overall expenses.
- Low Power design: The Automate 4.0 Reference Design is optimized for energy efficiency, which utilizes the Avant and Lattice Nexus™ FPGA platforms that use the power optimized LUT-4 (Look-Up Table) architecture.

In summary, the Automate 4.0 reference design offers an easy-to-use, cost-effective, and low power platform for developing advanced industrial solutions. Its comprehensive set of components and pre-configured IP ensure that customers can quickly and efficiently implement their projects, leading to faster time-to-market and reduced development costs.

1.3. Automate 4.0 Components

The Automate Stack 4.0 release includes the following components:

- System on Chip (SOC)
    - Main System IPs
        - EtherConnect IP (with RGMII, FIFO DMA, CNN Co-Processor Unit (CCU), SPI Flash Controller, Multiport extension,TSE MAC, and Reset Synchronizer.
    - Node System IPs
        - EtherConnect IP (With SGMII/RGMII (PHY or SFP), FIFO DMA, BLDC motor control IP, Data collector for predictive maintenance
        - Modbus, I2C 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). TCP/IP Ethernet stack is also added
    - 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

The overall architecture is shown in Figure 2.1. The Automate stack 4.0 consists of one Main System (MS) and multiple Node Systems (NS) (maximum eight in a chain). The host is connected to the MS through ethernet cable. Application software with user interface running on the 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 OPCUA 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 the main system, the Avant-E device is used for the demo design. For the node system, the Certus-NX Versa board is used for demo design.

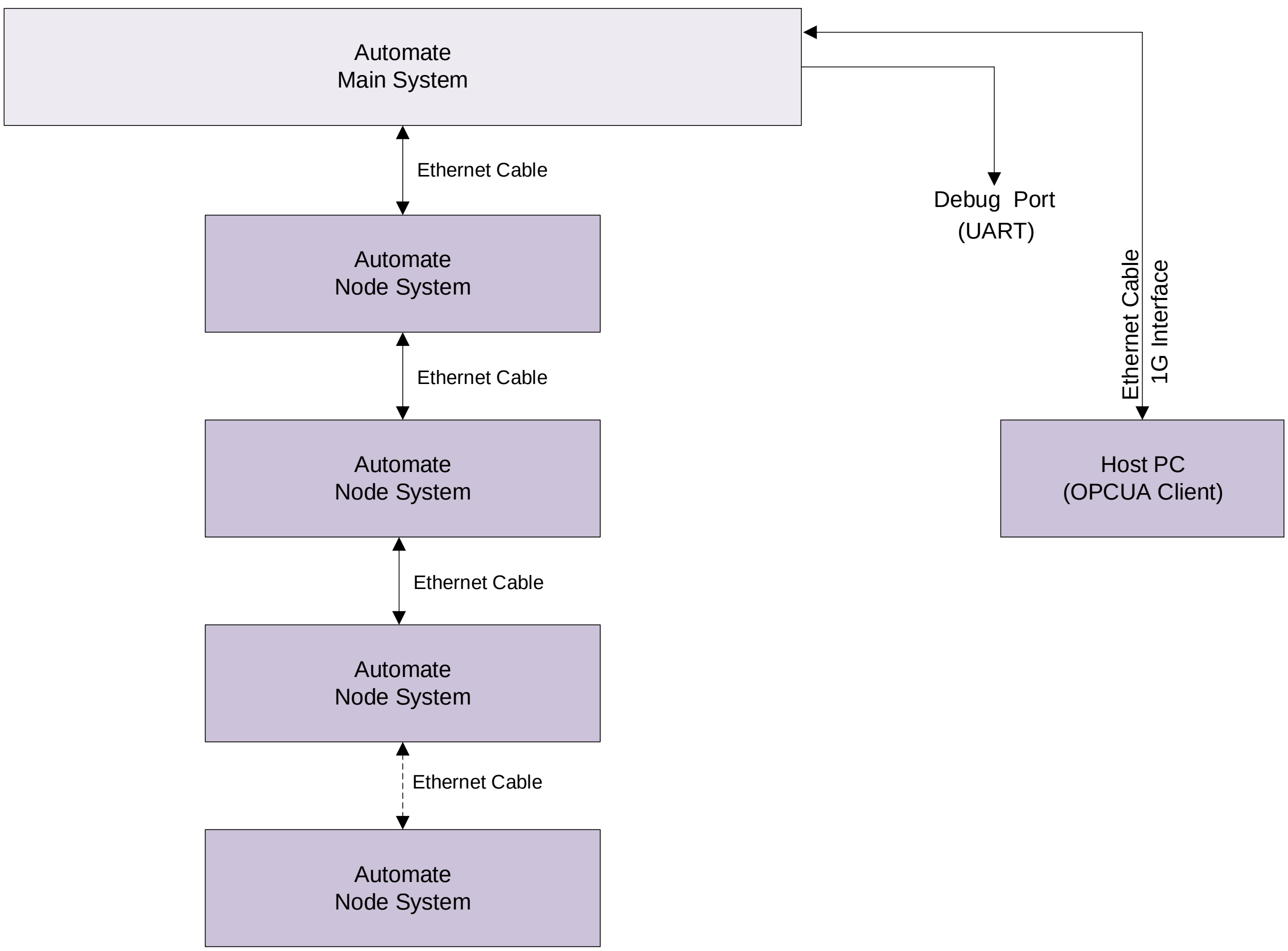


Figure 2.1. Lattice Automate Stack 4.0 Top Level Block Diagram

2.2. FPGA Design

2.2.1. Main System

The Main System is a System on Chip (SoC) designed for industrial automotive applications. It is integrated with several built-in Lattice Propel™ IP components, including a UART Controller that facilitates serial communication, QSPI Flash GPIO that manages general-purpose input/output operations for QSPI flash memory, LPDDR that supports low-power double data rate memory, and TSEMAC that provides triple-speed Ethernet MAC capabilities and the Scatter-Gather DMA (SGDMA) enables high-performance data transfers between IPs, eliminating the need for active CPU intervention and thus, improving overall system performance.

To further support industrial automotive applications, additional Intellectual Property (IP) components are integrated as part of the Automate Stack IP. These include a CNN Accelerator that boosts performance for convolutional neural networks, which is essential for advanced AI and machine learning tasks, FIFO DMA that ensures efficient data transfer between system components using First-In-First-Out Direct Memory Access, and EtherConnect IP that adds advanced Ethernet control features for improved network communication and management. The Automate Main System delivers a powerful and flexible platform tailored for industrial automotive applications, ensuring high performance and reliability. The Main System architecture is shown in Figure 2.2.

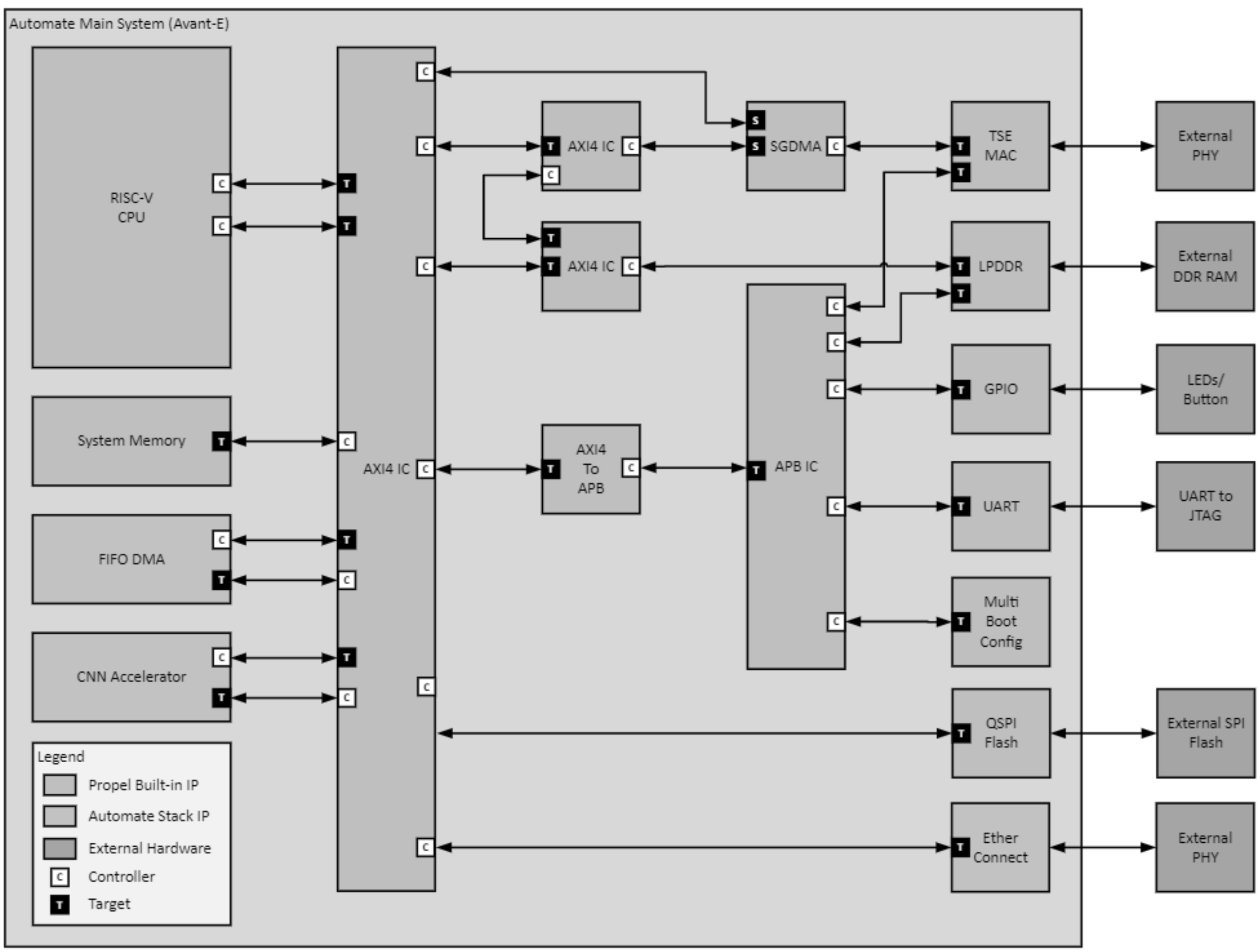


Figure 2.2. Automate 4.0 Main System Architecture

2.2.2. Lattice Main System 4.0 Architecture

This section describes architecture, dataflow details, and memory map address of the Lattice Automate Main System 4.0.

2.2.2.1. Lattice Main System 4.0 Architecture

The Main System architecture is shown in Figure 2.2. The AXI Interconnect has four controllers and eight targets.

- Four Controllers: RISC-V RX CPU Instruction Port, RISC-V RX CPU Data Port, FIFO DMA and CNN Co-processor
- Eight Targets: System memory, EtherConnect, FIFO DMA, CNN Co-processor, AXI2APB Bridge, SGDMA, and SPI Flash Controller

The RISC-V RX CPU, DCFIFO DMA and CNN Co-processor can access data to the shared memory Data Ram, SPI Flash Controller, EtherConnect, FIFO DMA, CNN Co-processor, and AXI2APB bridge directly and UART, TSE MAC, memory controller, FPGA Config module, and GPIO through AXI2APB bridge. The UART, EtherConnect, and GPIO can generate interrupts to RISC-V CPU.

2.2.2.2. Data Flow Details of the Main System 4.0

Automate Main System Multiboot Flow

The Avant-E 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 Avant-E device boots by loading the Primary pattern from the internal or external Flash. If loading of the Primary pattern fails, the Avant-E device attempts to load the Golden pattern. When a reprogramming of the bitstream is triggered through the toggling of the PROGRAMN pin or receiving a REFRESH command, 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.

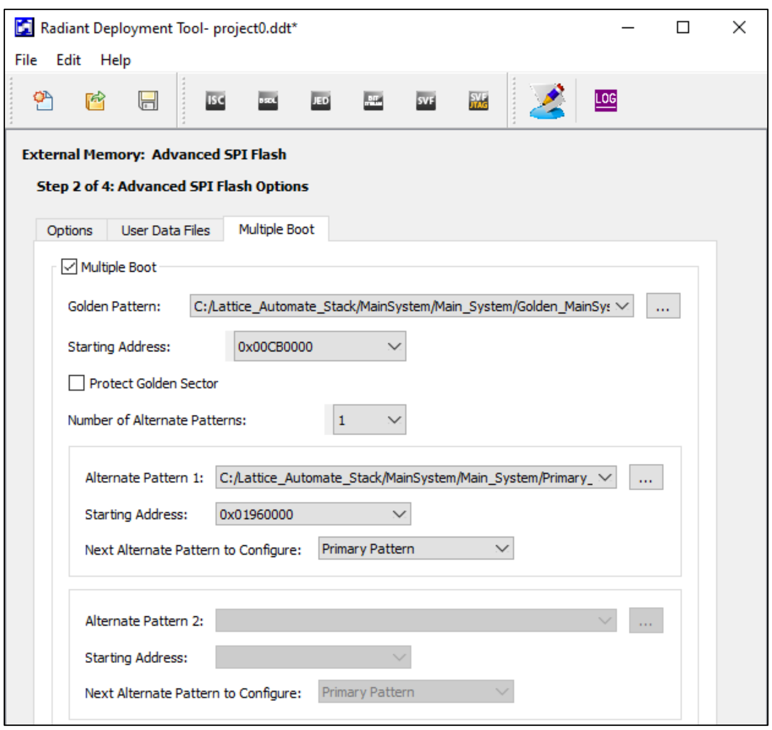


Figure 2.3. Multiboot Tab of Deployment Tool

Automate Main System Bootloader Flow

The Automate 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 baseline version of system. The primary image is an updated version of the system. The boot loader firmware supports CRC checking and switching between the primary Image and Golden image. The Firmware has the option to manually boot FPGA image based on CRC check.

Upon performing CRC check on the binary file, if the primary binary got corrupted somehow, the booting occurs from the golden one, but the bit file also must switch to golden. So, there is the firmware code in flash to switch the bit file to golden. And the same happens when primary bit file got corrupted. That means booting is done from one of the two sets of binary and bit file, firstly from primary and then from golden if the CRC check fails for 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 LPDDR4 memory controller to write the instruction code to a specific DDR4 memory location.

Automate Main System Application Flow

The DDR4 memory is divided into two parts, one for the instruction code for booting and the other like it was used in Automate 4.0 for buffering incoming and outgoing packets.

The SGDMA IP is used as data mover. It converts incoming UDP datagram from user application into AXI4 data and sends to LPDDR and similarly it converts AXI4 data coming from LPDDR and send it to the user application network stack, which basically does the data transfer between standard protocols.

RISC-V RX CPU can set the registers inside CNN Co-processor Unit (CCU) and start PDM operation. The CPU can poll another register in CNN Co-processor Unit (CCU) to check its operation status. RISC-V RX can request for the new data for predictive maintenance from node PDM data collector by sending instruction though EtherConnect IP.

The PDM data received from node through EtherConnect IP is transferred to data memory with DMA operation using FIFO DMA block or is sent to host directly through Ethernet through the LPDDR4 using AXI IP and TSE MAC

For the motor control, the commands from the host PC (OPCUA Client) are received in the OPCUA Server running on RISC-V RX CPU. The RISC-V RX CPU parses the command and sends the data to EtherConnect, which performs the packetization and send to downstream Node Systems. The RISC-V CPU can gather predictive maintenance data from downstream Node Systems through EtherConnect and send to the host through Ethernet.

The CPU can read data from EtherConnect through its AXI subordinate port, perform data processing, store the data at Data Ram, and then send to host. 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 RISC-V, an update is sent to the host through Ethernet.

RISC-V RX can also communicate with various peripherals connected to nodes through the SPI/I2C/UART interfaces other than motor through host commands. The data flow from OPCUA Client (Host PC) to OPCUA Server (Main board) and vice versa is shown below in Figure 2.4.

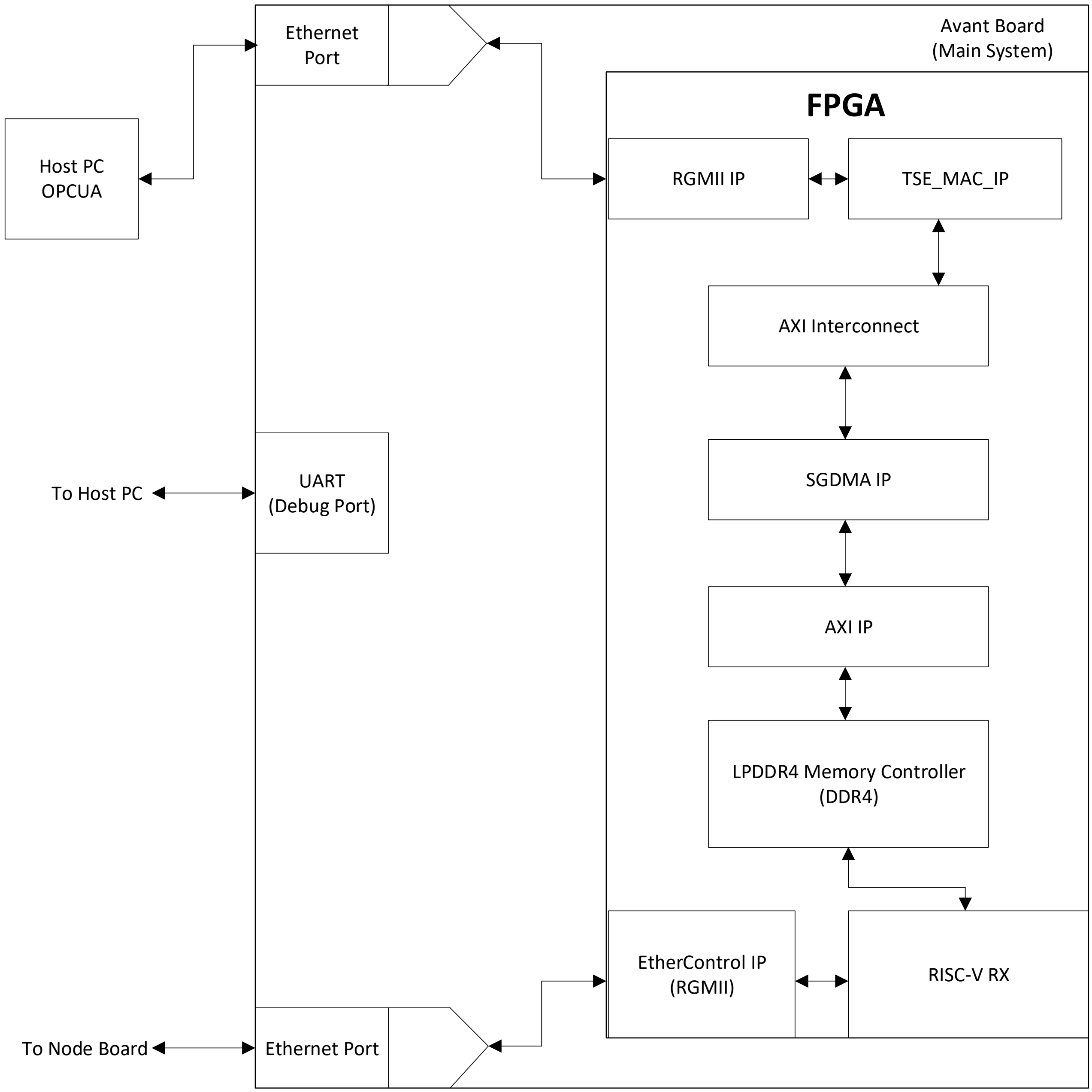


Figure 2.4. Client to Server Data Flow

2.2.2.3. Memory Map

The memory map of Main System is shown in Table 2.1.

Table 2.1.Main System Memory Map

Base Address	End Address	Range(bytes)	Block
0x40300000	0x40300FFF	4K	SPI FLASH CONTROLLER
0x00000000	0x000FFFFF	1M	System Memory
0x40000000	0x40000FFF	1K	GPIO
0x40001000	0x40004FFF	16K	TSE MAC
0x40090000	0x400903FF	1K	UART
0x40092000	0x40092FFF	4K	LPDDR4 Mem Controller APB
0x40098000	0x40098FFF	4K	SGDMA
0x40097000	0x40097FFF	4K	FPGA CONFIG APB
0x40310000	0x40317FFF	32K	FIFO DMA
0x40308000	0x4030FFFF	32K	EtherConnect
0x40318000	0x40318FFF	4K	CNN co-processor
0x80000000	0xBFFFFFFF	1G	LPDDR4 AXI
F2000000	F20FFFFF	1M	CLINT (CPU)
FC000000	FC3FFFFF	4M	PLIC (CPU)
F0000400	FFFFFFFF	250M	RESERVED (CPU)

2.2.3. Node System

The Node System architecture, shown in Figure 2.5, is same as the previous version. However, there is a new Encoder Subsystem been introduced. In addition, the Motor Control and PDM Data Collector has been enhanced with the capability to support closed loop feedback system where the motor positions are received from external EnDat Rotary Encoder periodically for motor speed control during runtime.

The AHBL Interconnect with three target interfaces and 10 controller interfaces connecting to respective IPs, namely:

- AHBL Target Interfaces
  - RISC-V CPU Instruction Cache
  - RISC-V CPU Data Cache
  - FIFO DMA
- AHBL Controller Interfaces
  - ISR RAM
  - Data Ram (S0 and S1)
  - Motor Control and PDM Data Collector (S0 and S1)
  - FIFO DMA
  - EtherConnect
  - SPI Flash Controller with Prefetch Buffer
  - AHBL2APB bridge
  - Encoder Subsystem

APB Interconnect has five controller interfaces and one target interface connecting to respective IPs, namely:

- APB Target Interfaces
  - AHBL to APB Bridge
- APB Controller Interfaces
  - GPIO
  - I2C
  - SPI
  - UART (Modbus)
  - Encoder Subsystem

Refer to Appendix A. Predictive Maintenance with TensorFlow Lite to see the data flow and memory map of the node system.

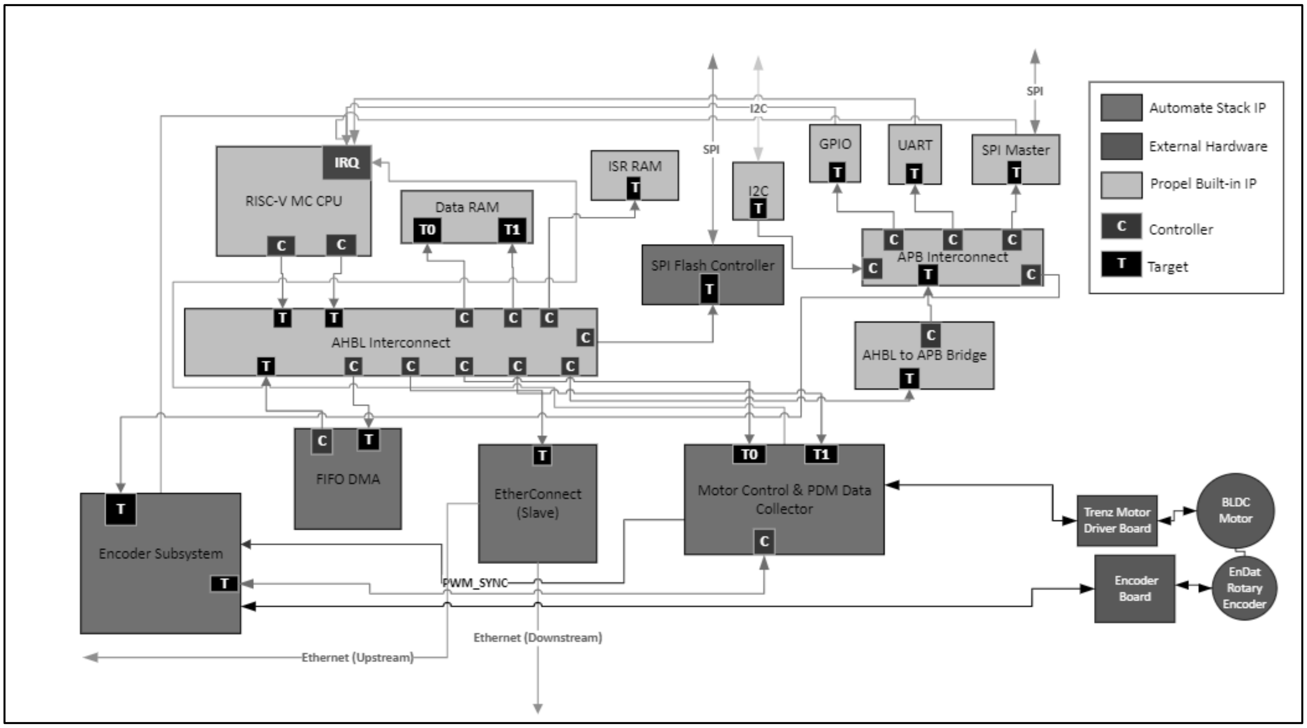


Figure 2.5. Node System Architecture

The Encoder Subsystem consists of the following components:

- APB Interconnect with two targets and one controller.
- SPI Controller IP where APB target interface is connected to the APB Interconnect and SPI controller interface is connected to the EnDat2.2 Master IP.
- EnDat2.2 Master IP where the target is connected to the controller through the SPI interface of the SPI Controller IP and EnDat interface is exported out from FPGA to external EnDat Rotary Encoder.

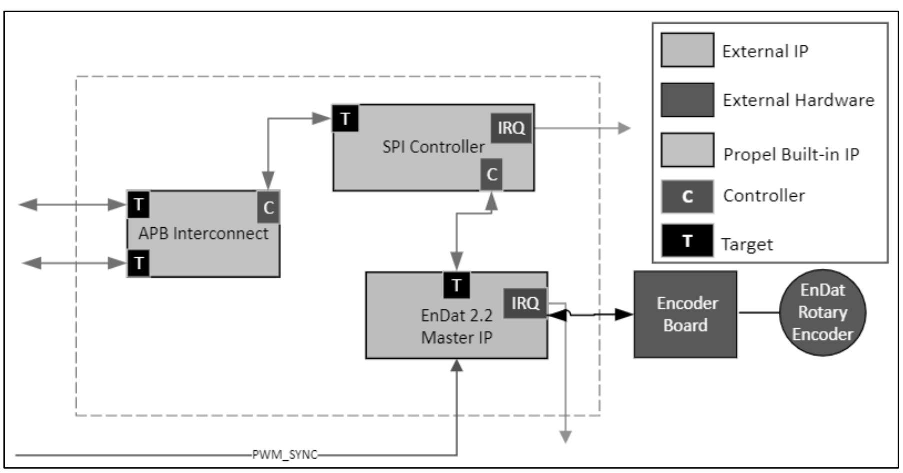


Figure 2.6. Encoder Subsystem Architecture

2.2.3.1. Data Flow

The RISC-V CPU stream its firmware from external SPI Flash through the SPI Flash Controller. The CPU can also access data to ISR RAM, Data RAM, access the register file inside EtherConnect, and control the registers at FIFO DMA and SPI Flash Controller. Either RISC-V CPU or FIFO DMA can move the data stored at the register file inside EtherConnect to Motor Control block. The RISC-V CPU or FIFO DMA can also move the data collected by PDM Data Collector back to EtherConnect and send out through Ethernet upstream port.

In addition, the firmware is also responsible to initialize the external EnDat encoder through communication through SPI Controller and EnDat2.2 Master upon power-up.

2.2.3.2. Memory Map

The Node System memory map is defined in Table 2.2.

Table 2.2. Node System Memory Map

Base Address	End Address	Range (Bytes)	Range (Bytes in hex)	Size (Kbytes)	Block
0x80000	0x807FF	2048	800	2	CPU PIC TIMER
0x190000	0x191FFF	8192	2000	8	CPU Instruction RAM
0x100000	0x107FFF	32768	8000	32	FIFO DMA
0x186C00	0x186FFF	1024	400	1	SPI Controller (Encoder Subsystem)
0x108000	0x10FFFF	32768	8000	32	EtherConnect
0x184800	0x184BFF	1024	400	1	GPIO
0x186000	0x1863FF	1024	400	1	I2C Master
0x184000	0x1843FF	1024	400	1	Motor Control and PDM Data Collector Port S0
0x185000	0x185FFF	4096	1000	4	Motor Control and PDM Data Collector Port S1
0x0	0x7FFFF	524288	80000	512	SPI Flash Controller
0x186800	0x186BFF	1024	400	1	SPI Master
0xC0000	0xCFFFF	65536	10000	64	CPU Data Ram Port S0
0xE0000	0xEFFFF	65536	10000	64	CPU Data Ram Port S1
0x186400	0x1867FF	1024	400	1	UART
0x80800	0xBFFFF	197632	30400	193	RESERVED
0xD0000	0xDFFFF	65536	10000	64	RESERVED
0xF0000	0xFFFFF	65536	10000	64	RESERVED
0x110000	0x183FFF	468992	74000	458	RESERVED
0x184400	0x1847FF	1024	400	1	RESERVED
0x184C00	0x184FFF	1024	400	1	RESERVED

2.3. EtherConnect IP Design Details

2.3.1. Overview of Existing IP

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

The EtherConect block can be used as a manager as well as a node based on the SYSTEM_TYPE parameter.

As a manager, EtherConnect IP has the output FIFO interface to send bulk data to DMA FIFO block and as node, it has the input FIFO interface to receive bulk data from DMA FIFO module.

As a manager, it works in four layers, such as AHBL layer, which is used to have connection with the RISC V CPU and register interface; application layer, which consists of data generation and sampling layers for the application; protocol layer, which is used to transmit and receive EtherConnect packets. Lastly, the physical layer transfer data with protocol layer in GMII protocol standard and it has RGMII and SGMII blocks to transmit or receive data over physical channels in RGMII or SGMII format.

The frame structure on 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: Node Configuration
- Packet type 02: Node Status
- Packet type 03: PDM Data Fetching

For Configuration type packet, the data written in FIFO present in application layer is as follows: the first four bytes indicate the packet type. The next four bytes indicate the node address. After that, the data is sent in the next four bytes. The subsequent content of the packet is dummy data (00) for 52 bytes or in a generalized case: (NODE_DATA_LENGTH - 12).

For Status type packet, the data written in FIFO present in application layer is as follows: the first four bytes indicate the packet type. The next four 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 status packet is 32-bit status value, which is fetched from a register (CH1_BASE_ADDR + 0x100).

For PDM type packet, the data written in FIFO present in application layer is as follows: the first four bytes indicate the packet type. The next four bytes indicate the node address. After that, the data is sent in the next four bytes. The next four 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

The packet communication remained the same as the previous released version. The request packets from the RISC-V CPU passes to the node system through the main system connection while the response of the status packets is written in a FIFO, which can be read by RISC-V CPU using the register BASE_ADDR + 0x2C.

2.3.2.1. Main System

The protocol layer and physical layer remains 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. The 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

At Node System, the FIFO is used to store complete sampled data of both configuration packets and status packets. Each node samples its own data only.

For the configuration packet, an interrupt is generated to indicate that the configuration is applied to the targeted peripherals (motor, I2C, and SPI) at the targeted node.

For status packet, the status of the targeted peripherals (motor, I2C, and SPI) of the targeted node are stored in the FIFO and the signal is generated that complete packet has been received in the FIFO and is ready to send response.

2.3.3. Register Map

The register map of the EtherConnect IP remains the same, except that one register is added to read the response of status the packet, which is highlighted in Table 2.3 and one register (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 2.3. EtherConnect IP Global Registers

EtherConnect Register Name	Register Function	Base Address (0x40308000)	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 2.4. EtherConnect IP Chain 1 Registers

EtherConnect Register Name	Register Function	Base Address (0x40308100)	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
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 EtherConnect manager Bus 0. The other interface is active for node and has the PDM data from the motor control data collector block. It has a Subordinate and a Manager interface where the Main System is in AXI4 interface, while the Node System is in AHBL interface. The register space for this block is as shown in Table 2.5.

The Subordinate interface is used to control DMA operations by external manager (which is CPU) and the Manager interface is used to perform for DMA operations.

Table 2.5. 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 2.6. 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 2.7. 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 2.8. 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 2.9. 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
000 = Disabled.
001 = Busy
010 = Done
100 = Error
Others = Reserved
STATUS[3:31]: Reserved

Table 2.10. 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
000 = Disabled.
001 = Busy
010 = Done
100 = Error
Others = Reserved
STATUS_RD[3:31]: Reserved

2.5. LPDDR4 Controller

An LPDDR (Low Power Double Data Rate) controller is a specialized memory controller designed to interface with LPDDR memory devices, which are widely used in mobile and embedded systems due to their low power consumption and high performance. The controller manages data transfers between the processor and LPDDR memory, ensuring efficient communication and optimal performance. In this system, LPDDR is used to store RISC-V program code and data.

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

2.6. 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. In Main system, QSPI is used to read main application from the SPI Flash.

For more information about the IP core including register map information, refer to QSPI Flash Controller IP User Guide (FPGA-IPUG-02248).

2.7. Scatter Gather DMA IP Design Details

A Scatter-Gather Direct Memory Access (SGDMA) controller is a specialized DMA engine designed to handle data transfers between memory and peripherals efficiently. It supports scatter-gather operations, which allow data to be transferred in non-contiguous blocks, improving flexibility and performance. In Main system, SGDMA is used to autonomously handle data transfer of LPDDR to and from TSE MAC with minimum interaction by the CPU.

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

2.8. CNN Co-Processor Unit (CCU)

The CNN Co-Processor Unit (CCU) is used to accelerate inference process for Predictive Maintenance in the main system.

For more details, refer to CNN Co-Processor Accelerator IP User Guide.

2.9. Motor Control and PDM Data Collector

The Motor Control and PDM Data Collector block has two AHBL subordinate interfaces and one APB manager interface:

- AHBL_S0 Interface – access control to motor configuration and status registers for PWM channel output controlling to external motor driver board.
- AHBL_S1 Interface – access control to predictive maintenance control and status registers for predictive maintenance data collection from the motor.
- APB_M0 Interface – initiate position fetching & update operation to the Encoder Subsystem when the Node system is running in a closed loop system.

The Motor Control and PDM Data Collector block is capable to be run in both open loop and closed loop system based on the input control ports exposed on the top level. This block is only available in the Node System. The captured data is sent to the Main System and processed by the CNN Co-processor unit mentioned in the CNN Co-Processor Unit (CCU) section. The steps to train the CNN model is further described in Appendix A.

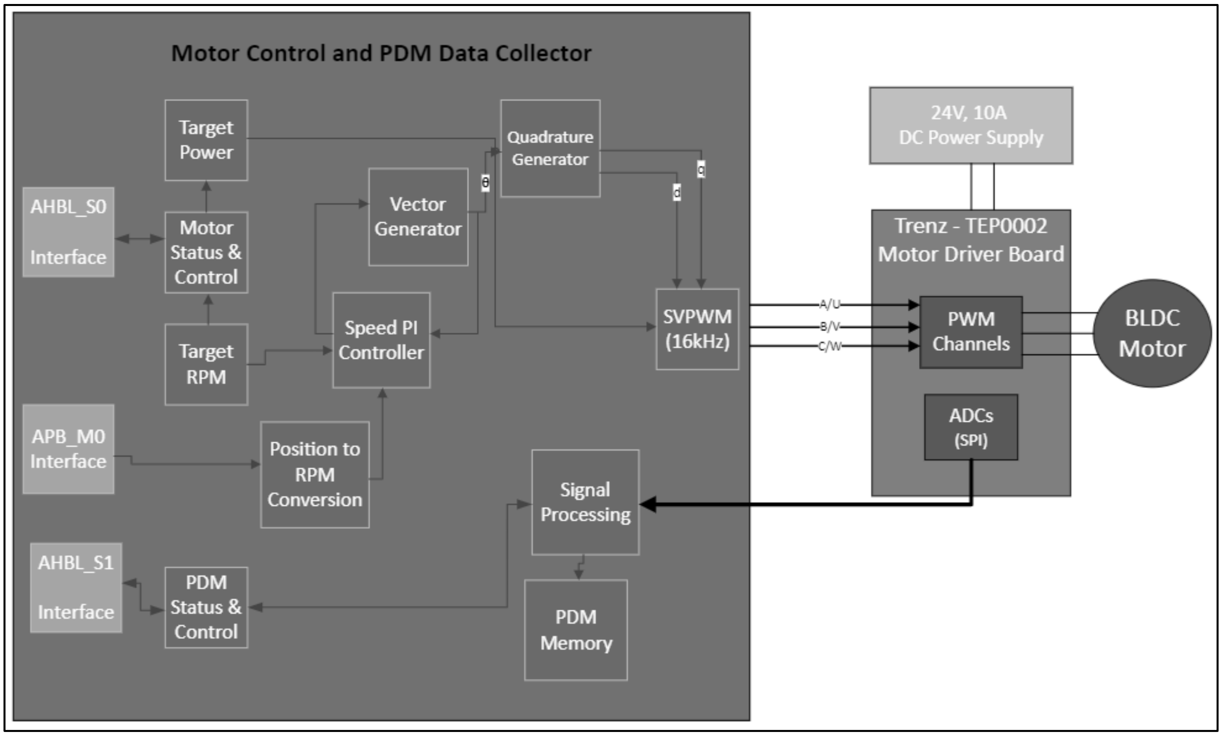


Figure 2.8. Motor Control and PDM Data Collector

The configuration and status registers accessible through the AHBL_S0 Interface and AHBL_S1 Interface are described in Table 2.11.

Table 2.11. Motor Control and PDM Data Collector Registers

Register Name	Register Function	Address (AHBL_S0 Base – 0x184000) (AHBL_S1 Base – 0x185000)	Access	Access Point	Reset Value
MTRCR0	Motor Control Register 0 – Min RPM	Base + 0x00	Read/Write	AHBL_S0	0x0
MTRCR1	Motor Control Register 1 – Max RPM	Base + 0x04	Read/Write	AHBL_S0	0x0
MTRCR2	Motor Control Register 2 – RPM PI KI	Base + 0x08	Read/Write	AHBL_S0	0x0
MTRCR3	Motor Control Register 3 – RPM PI KP	Base + 0x0C	Read/Write	AHBL_S0	0x0
MTRCR4	Reserved	Base + 0x10	Read	AHBL_S0	0x0
MTRCR5	Reserved	Base + 0x14	Read	AHBL_S0	0x0
MTRCR6	Motor Control Register 6 – Sync Delay and Control	Base + 0x18	Read/Write	AHBL_S0	0x0
MTRCR7	Motor Control Register 7 – Target RPM	Base + 0x1C	Read/Write	AHBL_S0	0x000A0000
MTRCR8	Reserved	Base + 0x20	Read/Write	AHBL_S0	0x0
MTRCR9	Reserved	Base + 0x24	Read/Write	AHBL_S0	0x0
MTRSR0	Motor Status Register 0 - RPM	Base + 0x28	Read	AHBL_S0	0x0
MTRSR1	Motor Status Register 1 – Limit SW and System Status	Base + 0x2C	Read	AHBL_S0	0x0
PDMCR0	Predictive Maintenance Control Register 0	Base + 0x30	Read/Write	AHBL_S0	0x0
PDMCR1	Predictive Maintenance Control Register 1	Base + 0x34	Read/Write	AHBL_S0	0x0
PDMSR	Predictive Maintenance Status Register	Base + 0x38	Read	AHBL_S0	0x0

Register Name	Register Function	Address (AHBL_S0 Base – 0x184000) (AHBL_S1 Base – 0x185000)	Access	Access Point	Reset Value
PDMDDR	Predictive Maintenance ADC Data Register	Base + 0x3C	Read	AHBL_S1	0x0
PDMQDR	Predictive Maintenance ADC Data Register	Base + 0x40	Read	AHBL_S1	0x0
BRDSW	DIP and Push Button Switches	Base + 0x50	Read	AHBL_S0	0x0
BRDLEDS	LEDs and 7-Segment	Base + 0x54	Read/Write	AHBL_S0	0xFFFFFFFF
Reserved	Reserved	Base + 0x58	Read	N/A	N/A
Reserved	Reserved	Base + 0x5C	Read	N/A	N/A
ENC_POS	Encoder Position	Base + 0x60	Read	AHBL_S0	0x0
Reserved	Reserved	Base + 0x64	Read	N/A	N/A
PWM_SYNC_IRQ	PWM_SYNC IRQ Status	Base + 0x68	Read/Write	AHBL_S0	0x0
Reserved	Reserved	Base + 0x6C	Read	N/A	N/A
Reserved	Reserved	Base + 0x70	Read	N/A	N/A
Reserved	Reserved	Base + 0x74	Read	N/A	N/A

Table 2.12. Motor Control 0 – Minimum RPM

MTRCR0				Base + 0x00
Byte	3	2	1	0
Name	RPM_PI_DELAY	MTRPOLES	Reserved	MINPWR
Default	0	0	0	0
Access	R/W			

MTRCR0[15:8]: Reserved
MTRCR0[7:0]: MINPWR – Minimum power for the initial open loop motor.
Note: The valid combination values of both TQ_PI_DELAY and MINPWR are 10 to (2^{16} -1).
MTRCR0[23:16]: MTRPOLES – Number of motor stator pole pairs. The value must be configured according to the datasheet for the specific motor. Valid values are up to 32 only.
MTRCR0[31:24]: RPM_PI_DELAY – Is the RPM PI update rate. Valid values are 1 to 255.

Table 2.13. Motor Control 1 – Maximum RPM

MTRCR1				Base + 0x04
Byte	3	2	1	0
Name	MAXAMPS	PWRGAIN	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[23:16]: PWRGAIN – Power gain for the initial open loop motor.
MTRCR1[31:24]: MAXAMPS – Breaker amps for the initial open loop motor.

Table 2.14. 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 – 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 – The Integrator Anti-Windup threshold. Valid values are 1 to (2^{16} -1).

Table 2.15. 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 – 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 – The Integrator Anti-Windup Clamp. Valid values are 1 to (2^{16} -1).

Table 2.16. 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 nanoseconds (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]: Reserved
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]: Reserved
MTRCR6[30]: DIRECTION – Direction of motor depending on the MTR_TYPE value.

Table 2.17. Direction Mapping

MTR_TYPE	Direction
0	0 = Clockwise Rotation, 1 = Counter-Clockwise Rotation
1	1 = Clockwise Rotation, 0 = Counter-Clockwise Rotation

MTRCR6[31]: ENGAGE – Sync Signal to latch all Control Registers from AHBL clock domain (50–100 MHz) to Motor clock domain (20 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 2.18. Motor Control Register 7 – Target RPM

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

MTRCR7[15:0]: TRGRPM – Target RPM. Valid values are 0 to (2^{16} -1).
MTRCR7 [16]: MTR_TYPE – The value of this bit determines the behavior of the value in the DIRECTION to be interpreted by the Motor Control IP.
Note: For Anaheim motor, this bit must be set to 0.
MTRCR7 [31:17]: Reserved

Table 2.19. Motor Status Register 0 – RPM

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

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

Table 2.20. 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[6]: I_LOOP_CONTROL
0 = Open Loop
1 = Close Loop
MTRSR1[7]: DRIVE_FAULT
0 = Drive fault not occurred.
1 = Drive fault occurred. This bit is coming from motor driver board that driving to the actual motor when overcurrent fault detected from protection circuit.
MTRSR1[8]: ECB_TRIPPED
0 = ECB tripped not occurred.
1 = ECB tripped occurred due to the feedback current received from motor driver board exceeded the value configured to MAXAMPS.
MTRSR1[10:9]: ENC_POS_BIT
2’b00 = Reserved
2’b01 = EnDat Encoder.
2’b10 - 2’b11 = Reserved
MTRSR1[30:11]: Reserved
MTRSR1[31]: ENC_LINK_STAT
0 = Encoder link is not established.
1 = Encoder link is established.

Table 2.21. 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 is 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 2.22. 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 2.23. 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]: Reserved
PDMSR [31:16]: PDMSR_ROT – Current count of Theta rotations PDM data has been collected for.

Table 2.24. 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[1]
PDMDDR [31:16]: ADC1 Voltage or Current reading Phase B[1]

Table 2.25. 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[1]
PDMQDR [31:16]: ADC3 Voltage or Current reading of DC supply[1]

Table 2.26. Versa Board Switch Status Register

BRDSW				Base + 0x50
Byte	3	2	1	0
Name	Reserved	Reserved	Reserved	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]: Reserved.
Bits [31:8]: Reserved.

Table 2.27. Versa Board LED and PMOD Control Register

BRDLEDS				Base + 0x54
Byte	3	2	1	0
Name	Reserved	Reserved	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
Bits [31:16]: Reserved.

Table 2.28. Encoder Position Register

PDMQDR				Base + 0x60
Byte	3	2	1	0
Name	ENC_POS			
Default	0	0	0	0
Access	R			

ENC_POS [31:0]: Motor position received from Encoder. The resolution depends on the specific Encoder model used. For Endat Encoder, it is up to resolution of 25 bits.

Table 2.29. PWM_SYNC IRQ Status Register

PDMQDR				Base + 0x60
Byte	3	2	1	0
Name	Reserved			PWM_SYNC_IRQ
Default	0	0	0	0
Access	R			

PWM_SYNC_IRQ [0]: IRQ status whenever PWM_SYNC is issued out from Motor Control and PDM Data Collector IP.
PWM_SYNC_IRQ [7:1]: Reserved.

2.10. SPI Controller IP

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

For the SPI controller IP within Encoder Subsystem, it is used to communicate with the third-party Encoder Master IP for data communication on the Encoder initialization and status monitoring purpose.

For more details, refer to SPI Controller IP User Guide (FPGA-IPUG-02069).

2.10.1. SPI Controller Register Map

For the register description, refer to the chapter 5 from SPI Controller IP User Guide (FPGA-IPUG-02069) for more details.

2.10.2. Programming Flow

2.10.2.1. Initialization

The following SPI Controller registers must be set properly before performing the SPI transaction:

- CHP_SEL_REG – Set 1'b1 to the bit for the corresponding target. Set 1'b0 to other bits.
- CHP_SEL_POL_REG – Can 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 host device needs to update the above registers only when SPI Controller is switching to different target device. No need to perform the initialization again if the next transaction is for the currently selected target device.

For more details, refer to SPI Controller IP User Guide (FPGA-IPUG-02069).

2.10.2.2. Transmit/Receive Operation

For more details on the general recommended operation flow, refer to SPI Controller IP User Guide (FPGA-IPUG-02069).

For the SPI controller IP within Encoder Subsystem, the following sequence is used for data communication to any register defined in the third-party Encoder Master IP during Encoder initialization stage:

1. Write to FIFO_RST_REG to assert reset on both TX and RX FIFOs in the SPI Controller.
2. Write to INT_STAT_REG to reset all interrupt status bits in the SPI Controller.
3. Write to FIFO_RST_REG to de-assert reset on both TX and RX FIFOs in the SPI Controller.
4. Write to WORD_CNT_RST_REG to reset the word count in the SPI Controller.
5. Write to TGT_WORD_CNT_REG according to the number of words to transfer in the SPI Controller.
6. Write n-word data to WR_DATA_REG, amounting to less than or equal to Transmit FIFO depth. If target n-word is greater than the Transmit FIFO depth, check the interrupt for Transmit FIFO full, INT_STATUS_REG.tx_fifo_full_int, before writing data to WR_DATA_REG to avoid data loss.
7. Clear the pending interrupts in INT_STATUS_REG as needed.
8. Read INT_STATUS_REG. Check if the pending interrupt is tr_cmp_int. This indicates that the SPI target has completed transmitting the target n-word data.
9. Clear the pending interrupt in INT_STATUS_REG.
10. If CFG_REG.only_write = 1'b0, read the n-word data in RD_DATA_REG.

    Note: Based on the third-party Encoder Master IP specification, two header bytes are required to be transmitted.

2.11. I2C Controller IP

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 must be minimized. The device that initiates the transmission on the I2C bus is commonly known as the Controller, while the device being addressed is called the Target. The I2C Controller 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.

For more information about the IP core including register map information, refer to I2C Controller IP User Guide (FPGA-IPUG-02071).

2.12. UART IP

The Universal Asynchronous Receiver/Transmitted (UART) Transceiver IP core performs serial-to-parallel conversion of data characters received from a peripheral UART device and parallel-to-serial conversion of data characters received from the host locater insider the FPGA through an APB interface. In this system, UART is usually connected to terminal character printing and debugging purpose.

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

2.13. EnDat 2.2 Master IP

The EnDat 2.2 Master IP handles the communication with EnDat Rotary Encoder. This simplifies the transmission of position data and additional data to the higher-level application.

The EnDat 2.2 Master IP consists of the following interfaces:

- EnDat interface that communicate to the external EnDat Rotary Encoder during initialization stage as well as the normal operation stage for control and monitoring.
- SPI interface for communication with SPI Controller where the initialization sequence is performed by CPU. During normal operation, the Motor Control and PDM Data Collector initiates the transaction through the SPI Controller periodically to retrieve encoder position values through receive registers as defined in the EnDat 2.2 Master IP.

For more details, refer to the representative through the Heidenhain website to inquire about EnDat 2.2 Master IP.

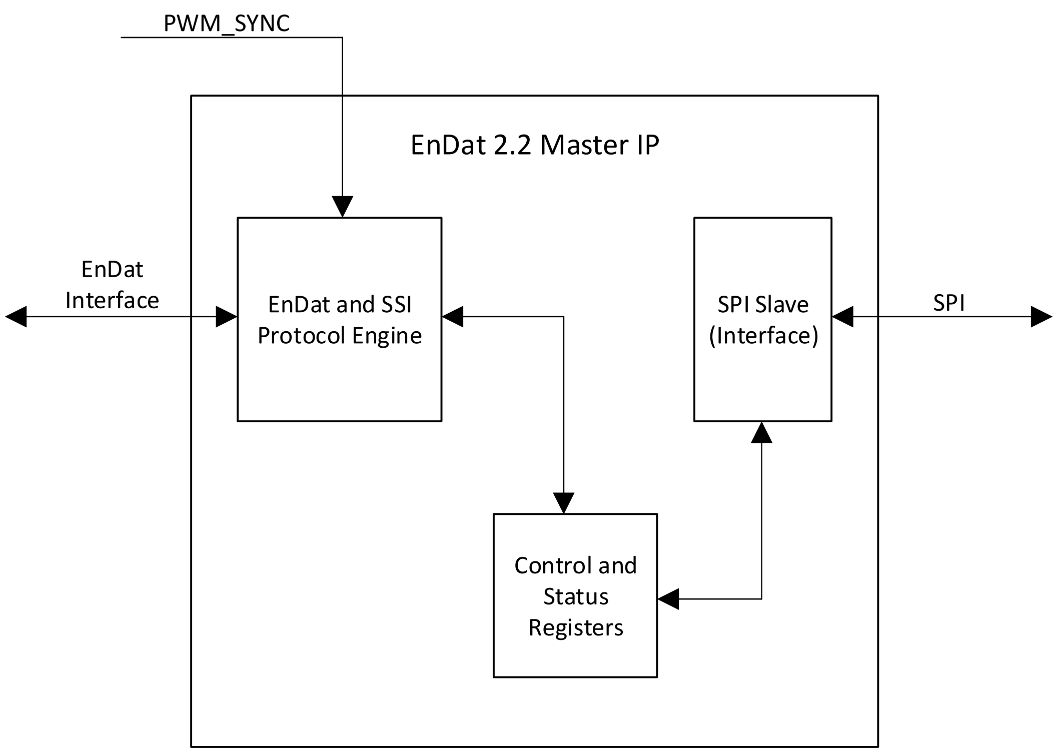


Figure 2.9. EnDat 2.2 Master IP Core Functional Block Diagram

2.14. SPI Flash Controller

The SPI Flash Controller is designed to stream data from external flash to FPGA using quad SPI data lines through execute-in-place (XiP) access. It has a prefetch buffer to enable cache feature for internal block of FPGA. This block does not have any configuration register for controlling as the basic settings (static configuration) are configured only during build generation. This block does not support flash data write operation as it is only used in the Node System SoC only for instruction streaming to RISC-V from external SPI flash. This block is only supporting Micron and Macronix currently.

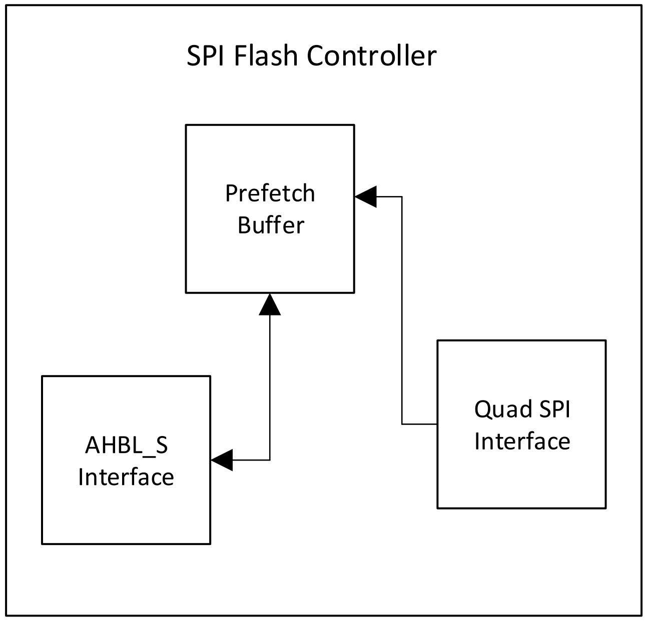


Figure 2.10. SPI Flash Controller IP Core Functional Block Diagram

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. In this system, TSEMAC is configured to RGMII mode and MDIO interface is used to control the external PHY control and status registers.

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

2.16. FPGA Config Module Design

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

3. Resource Utilization

The resource utilization for the Main System is shown in Table 3.1 and Table 3.2.

Table 3.1. Main System Resource Utilization

Blocks	LUT4 Logic	LUT4 Distributed RAM	LUT4 Ripple Logic	PFU Registers	I/O Registers	I/O Buffers	DSP MULT	EBR
soc_golden_gsrd	46600(9)	16224(0)	6696(0)	43618(2)	2(0)	107(46)	10(0)	205(0)
apb_interconnect0_inst	108(0)	0(0)	0(0)	6(0)	0(0)	0(0)	0(0)	0(0)
axi2apb0_inst	253(0)	0(0)	54(0)	198(0)	0(0)	0(0)	0(0)	0(0)
axi4_interconnect0_inst	11467(0)	6522(0)	712(0)	11308(0)	0(0)	0(0)	0(0)	0(0)
axi4_interconnect1_inst	2287(0)	1764(0)	90(0)	2955(0)	0(0)	0(0)	0(0)	0(0)
axi_register_slice0_inst	165(1)	0(0)	0(0)	307(0)	0(0)	0(0)	0(0)	0(0)
cnn_coproccesor0_inst	711(0)	0(0)	374(0)	1009(0)	0(0)	0(0)	4(0)	0(0)
cpu0_inst	4980(0)	252(0)	1262(0)	3404(0)	0(0)	0(0)	6(0)	15(0)
etherconnect0_inst	5109(0)	96(0)	894(0)	2944(0)	0(0)	0(0)	0(0)	17(0)
fifo_dma1_inst	543(0)	0(0)	294(0)	613(0)	0(0)	0(0)	0(0)	8(0)
gpio0_inst	115(0)	0(0)	0(0)	97(0)	0(0)	8(0)	0(0)	0(0)
lpddr4_mc_contr0_inst	7871(0)	1482(0)	1056(0)	8992(0)	0(0)	49(0)	0(0)	25(0)
mbconfig0_inst	14(0)	0(0)	0(0)	64(0)	1(0)	0(0)	0(0)	0(0)
mpmc0_inst	3027(0)	714(0)	350(0)	4173(0)	0(0)	0(0)	0(0)	18(0)
osc0_inst	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
pll0_inst	22(0)	0(0)	0(0)	15(0)	0(0)	0(0)	0(0)	0(0)
qspi0_inst	3104(0)	0(0)	346(0)	2149(0)	0(0)	4(0)	0(0)	0(0)
rst_sync0_inst	43(0)	0(0)	32(0)	36(0)	0(0)	0(0)	0(0)	0(0)
sgdma0_inst	1519(0)	0(0)	556(0)	2085(0)	0(0)	0(0)	0(0)	8(0)
sysmem0_inst	1248(0)	0(0)	180(0)	692(0)	0(0)	0(0)	0(0)	112(0)
tse_mac0_inst	2741(0)	3840(0)	412(0)	1842(0)	0(0)	0(0)	0(0)	2(0)
tse_to_rgmii_bridge0_inst	620(0)	1554(0)	36(0)	119(0)	0(0)	0(0)	0(0)	0(0)
uart0_inst	644(0)	0(0)	48(0)	608(0)	1(0)	0(0)	0(0)	0(0)

Table 3.2. Main System Total Resource Utilization

LUT4	70410
PFU Register	44520
I/O Buffers	85
EBR	157

The resource utilization for the Node System is shown in Table 3.3 and Table 3.4.

Table 3.3. Node System Resource Utilization

Blocks	LUT4 Logic	LUT4 Distributed RAM	LUT4 Ripple Logic	PFU Registers	I/O Registers	I/O Buffers	DSP MULT	EBR	Large RAM
soc_node_top	15818(2)	702(0)	4768(0)	13061(1)	19(4)	77(65)	27.5(0)	70(0)	1(0)
dut_inst	15816(1)	702(0)	4768(0)	13060(0)	15(0)	12(0)	27.5(0)	70(0)	1(0)
ISR_RAM_inst	50(0)	0(0)	0(0)	30(0)	0(0)	0(0)	0(0)	4(0)	0(0)
ahbl0_inst	188(0)	0(0)	0(0)	505(0)	0(0)	0(0)	0(0)	0(0)	0(0)
ahbl2apb0_inst	286(0)	0(0)	0(0)	190(0)	0(0)	0(0)	0(0)	0(0)	0(0)
apb0_inst	28(0)	0(0)	0(0)	8(0)	0(0)	0(0)	0(0)	0(0)	0(0)
cpu0_inst	2608(2)	0(0)	432(0)	1659(2)	0(0)	0(0)	0(0)	2(0)	0(0)
dma_fifo_inst	477(0)	0(0)	310(0)	545(0)	0(0)	0(0)	0(0)	16(0)	0(0)
encoder_subsys_inst	2971(0)	0(0)	480(0)	1914(0)	4(0)	0(0)	0(0)	2(0)	0(0)
ether_control_inst	3116(0)	288(0)	1282(0)	3053(0)	0(0)	6(0)	0(0)	27(0)	0(0)
gpio0_inst	64(0)	0(0)	0(0)	50(0)	3(0)	0(0)	0(0)	0(0)	0(0)
i2c_master0_inst	435(0)	24(0)	126(0)	506(0)	0(0)	2(0)	0(0)	0(0)	0(0)
motor_control_data_collector_inst	3164(0)	366(0)	2004(0)	3888(0)	3(0)	0(0)	27.5(0)	18(0)	0(0)
pll0_inst	21(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
spi_flash_controller0_inst	154(0)	0(0)	38(0)	191(0)	4(0)	4(0)	0(0)	1(0)	0(0)
spi_master0_inst	280(0)	24(0)	50(0)	307(0)	0(0)	0(0)	0(0)	0(0)	0(0)
sysmem0_inst	102(0)	0(0)	0(0)	68(0)	0(0)	0(0)	0(0)	0(0)	1(0)
uart0_inst	198(0)	0(0)	46(0)	146(0)	1(0)	1(0)	0(0)	0(0)	0(0)

Table 3.4.Node System Total Resource Utilization

LUT4	21288
PFU Register	13061
I/O Buffers	77
EBR	70

4. Firmware

4.1. Main System Boot Flow

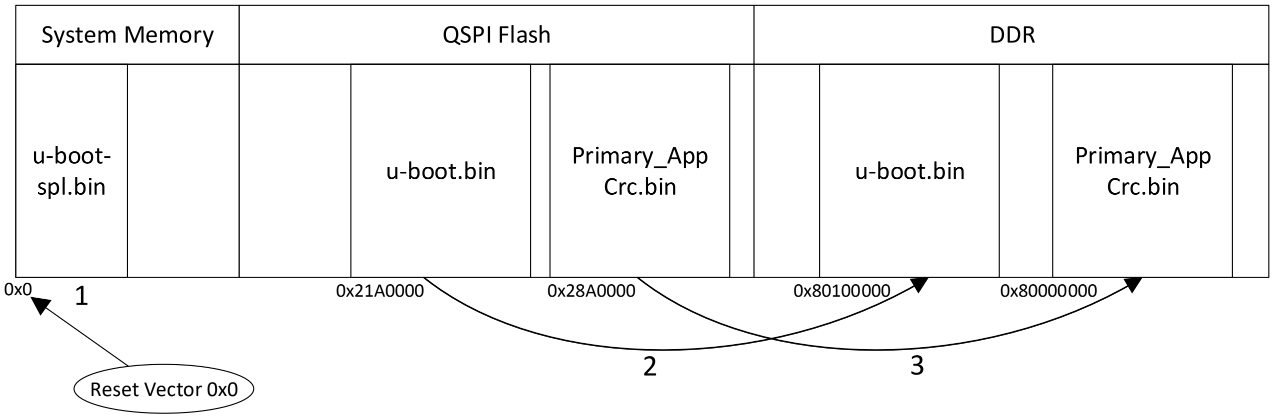


Figure 4.1. Main System Boot Flow

Below is the main system boot up sequence:

1. U-Boot SPL is run upon power up.
2. SPL copy U-Boot Proper from flash address 0x21A0000 to DDR address 0x80100000 and jump to 0x80100000.
3. U-Boot Proper copy FreeRTOS application from flash address 0x28A0000 to 0x80000000 and jump to 0x80000000.

4.2. Node System Boot Flow

There is no bootloader for node system. The node system runs the firmware from the SPI flash XIP. Refer to the Node System APIs for more information.

5. Software APIs

5.1. Main System

5.1.1. 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 IP. 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 particular 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.

Below are the available APIs for the operations:

- 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 Initialisation 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 chains 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. 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, uint32_t pdm_data_base_addr)
  The API is present in the main.c file. It needs one parameter:
  - total_size: the size of the PDM data required from user interface
  - node_addr: node select value sent in packet
  - pdm_data_base_addr: PDM base address

  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.
  The 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.

5.1.2. lwIP Ethernet and UDP stack

The Ethernet and UDP stack are based on lwIP stack. The implementation is ported into the FreeRTOS framework. The connection from the Main to Host user interface is managed by the lwIP stack communicating through the UDP protocol.

The Ethernet stack performs the following tasks:

- Receive – Polling ethernet data packet from the SGDMA Rx Buffer and forwards the packet to the higher software stack for processing the OPCUA data from the Host user interface
- Transmit – Sends the data from the OPCUA stack to the Host user interface

The UDP stack includes the following:

- ICMP – Respond and reply to ICMP queries from Host GUI to the Main system
- Addressing – Assigning IP address and MAC address to the Main system
- Payload – Decoding the payload from the Host GUI to the OPCUA and encapsulating the payload to the sent to the Host user interface.

5.1.3. OPCUA PubSub

In the PubSub model, a publisher component, which can define *DataSets* that contain Variables or EventNotifiers. The Publisher publishes DataSetMessages, which contain DataChanges or Events. The sender defines in Datasets what is sent, instead of the receiver. The 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.

The OPCUA defines two different network types for PubSub.

- Local Network – which can use UDP Broadcast (or Unicast in some cases) or Ethernet APL. The messages are optimized binary UADP, which is defined in the OPCUA specifications. Only the OPCUA subscribers can interpret the messages.
- Message Queue Broker – which 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 the following functions:

- Generic variable Create_UADP_NetwokMessage ()
- Generic variables UADP NetworkMessage_parse (**)**

5.1.4. Create_UADP_NetworkMessage

5.1.4.1. 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 must be omitted, if bit 7 of the UADPFlags is false. The PublisherId type is of DataType Uint16.
- ExtendedFlags2 – The ExtendedFlags2 must be omitted if bit 7 of the ExtendedFlags1 is false.
- PublisherId – The Id of the Publisher that sent the data. Valid DataType are Uintger (unsigned integer) and String.
- DataSetClassId – The DataSetClassId associated with the DataSets in the NetworkMessage.

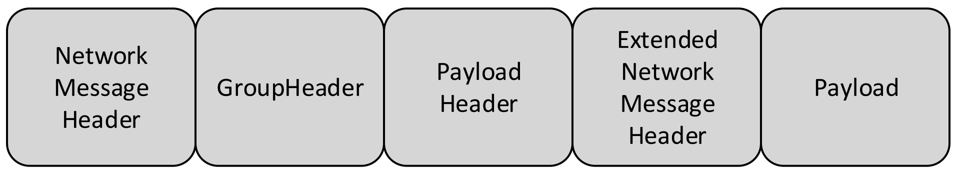


Figure 5.1. UADP Version

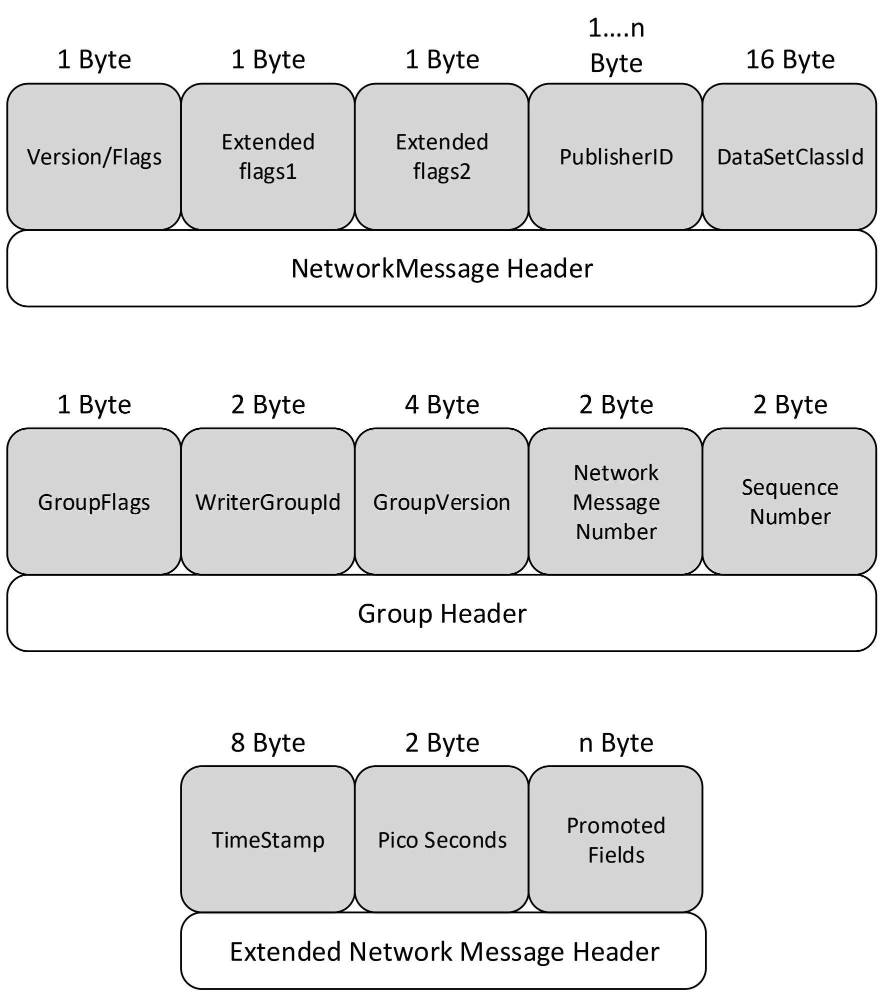


Figure 5.2. UADP Message Packet Header

5.1.5. GroupHeader

The group header must be 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.

5.1.6. Extended NetworkMessage Header

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

5.1.6.1. Payload

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

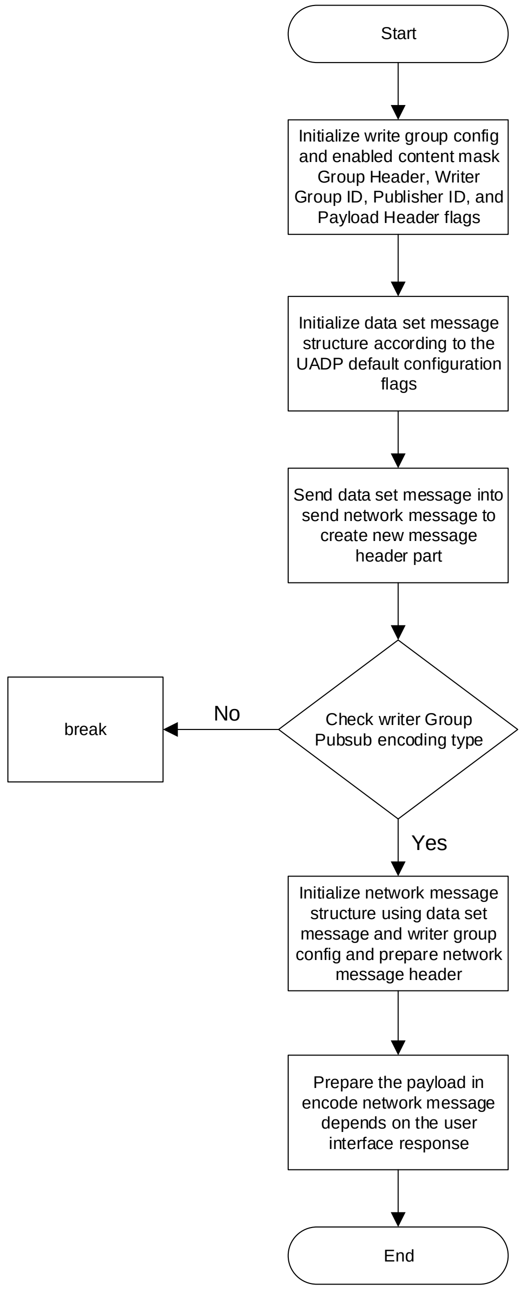


Figure 5.3. Create_UADP_NetworkMessage

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, the data gets the node IDs, and these node IDs identify the method call or node variables or method variables. After identification, create an UDP data response header, csv nodeid, request type and value, and write the UDP data request on LPDDR memory and get the UDP data response from LPDDR memory. Parse data get method nodeIds then called the method according to the method nodeId such as startmotor, stop motor, and power off.

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

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

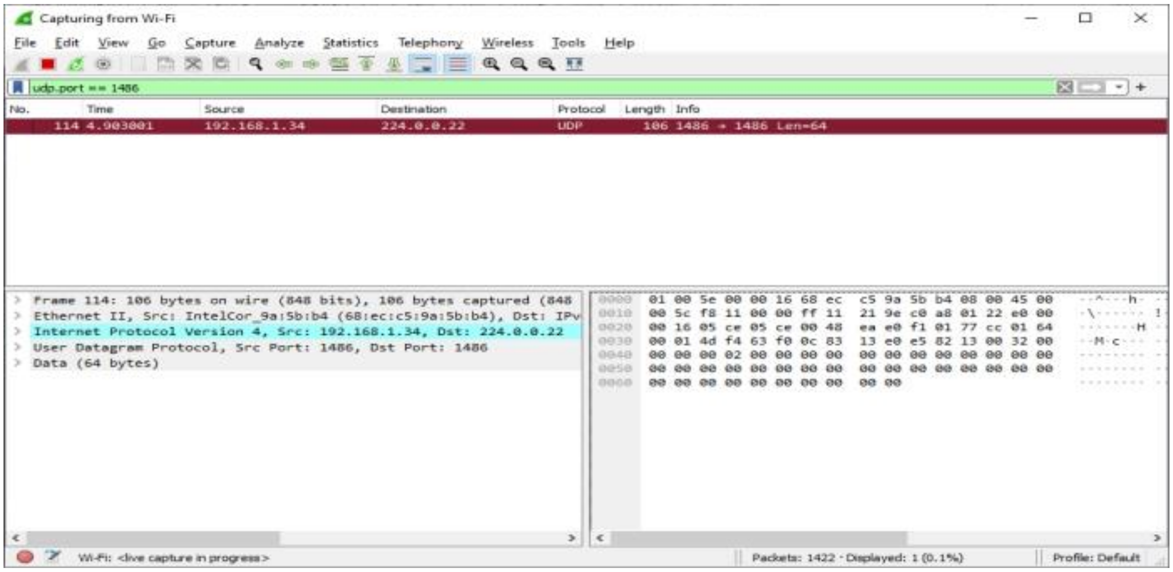


Figure 5.4. UADP Network Message Format

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 UADP_NetworkMessage.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, and max power.

```
void rfl_update_config()
```

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

stop_motor

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

```
void stop_motor()
```

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

poweroff_motor

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

```
void poweroff_motor()
```

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

get_background

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

```
void get_background()
```

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

run_pdm

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

```
void run_pdm();
```

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

5.2. Node System APIs

5.2.1. Tasks of the Node System

The Node System acts to control the Motor Control and PDM Data Collector 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 perform the following tasks:
  - Configures the peripherals (Motor Control, I2C, SPI, and Modbus).
  - Provides the status of the peripherals (Motor Control, I2C, SPI, and Modbus).
  - Provides the PDM data collected through Motor Control and PDM Data Collector.
- Perform key functions

5.2.2. Key Functions

- 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 peripherals 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 the external Encoder as well as Modbus, SPI, and I2C 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 peripherals (I2C, Modbus, SPI, and Motor Control) 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 peripherals (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
  - I2C Master
    The following are the I2C BSP functions used in the main.c file for writing and reading the I2C slave 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 Master
    The following are the SPI BSP functions used in the main.c file for writing and reading SPI slave 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 OPCUA header format. In this API, store the publisher ID and writer ID these IDs are used into 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 writer ID, publisher ID, and writer group ID and use of these IDs in the OPCUA pub-sub communication.
- Encoder init
  *uint8_t encoder_init(*encoder_id*)*
  This API is called to initialize the specific Encoder model according to the initialization sequence described in specification from third party vendor.

6. 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.

6.1. Communication between Host and Main System

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

6.1.1. 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

6.1.2. 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

6.2. 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.

6.2.1. 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

6.2.2. 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

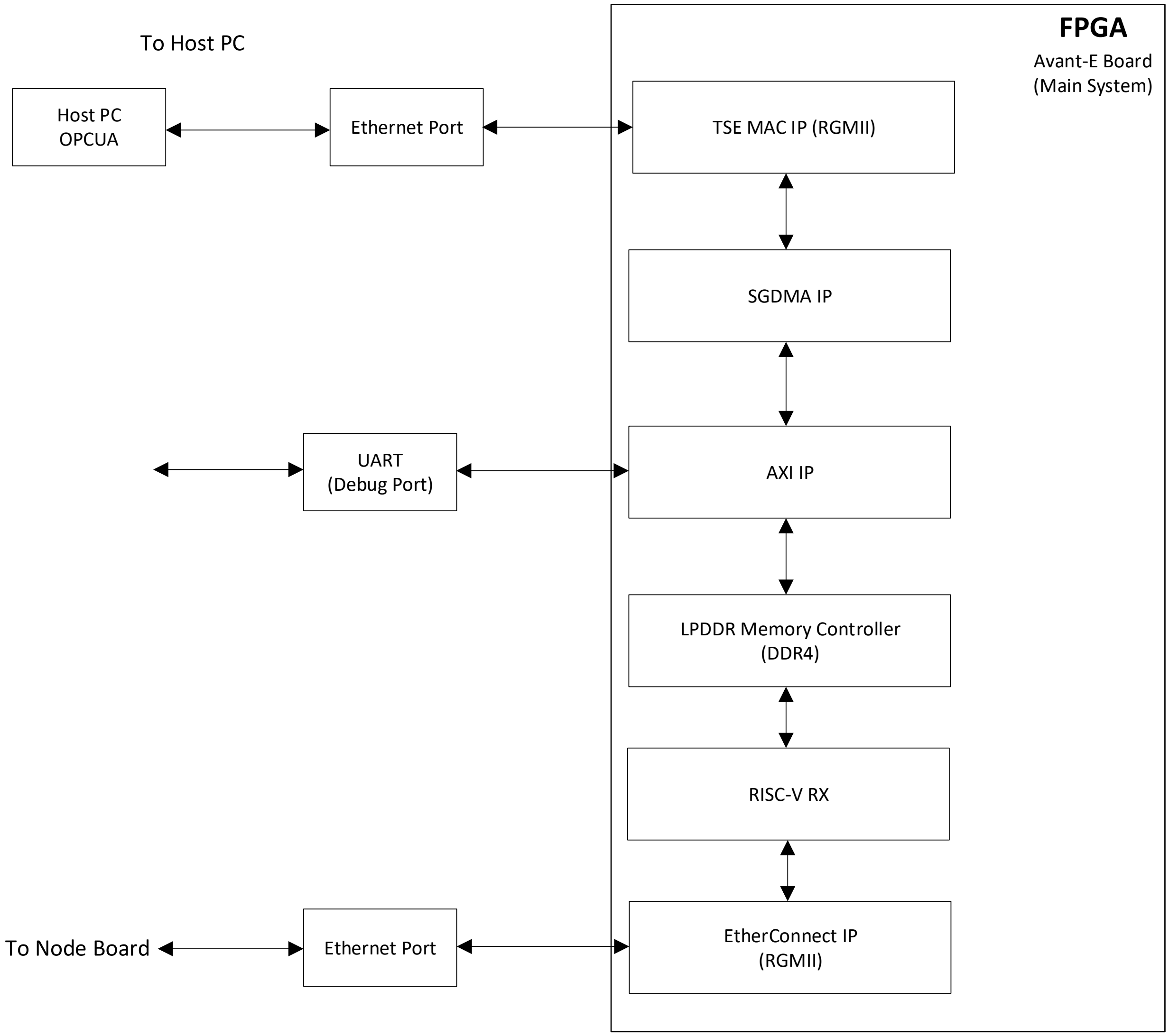


Figure 6.1. Data Flow from Host to Node System through the Main System

Appendix A. Predictive Maintenance with TensorFlow Lite

A.1. Overview

The Predictive Maintenance (PDM) section outlines the steps necessary (shown in Figure A.1) for rebuilding the model with your own data. It begins with the data capturing process, followed by the algorithm used to train the Convolutional Neural Network (CNN) model, which includes details on the neural network architecture, the training process, and model testing and accuracy. Finally, it covers the algorithm for running inference on the device, including the compilation of the TensorFlow Micro library and optimization for the CNN co-processor.

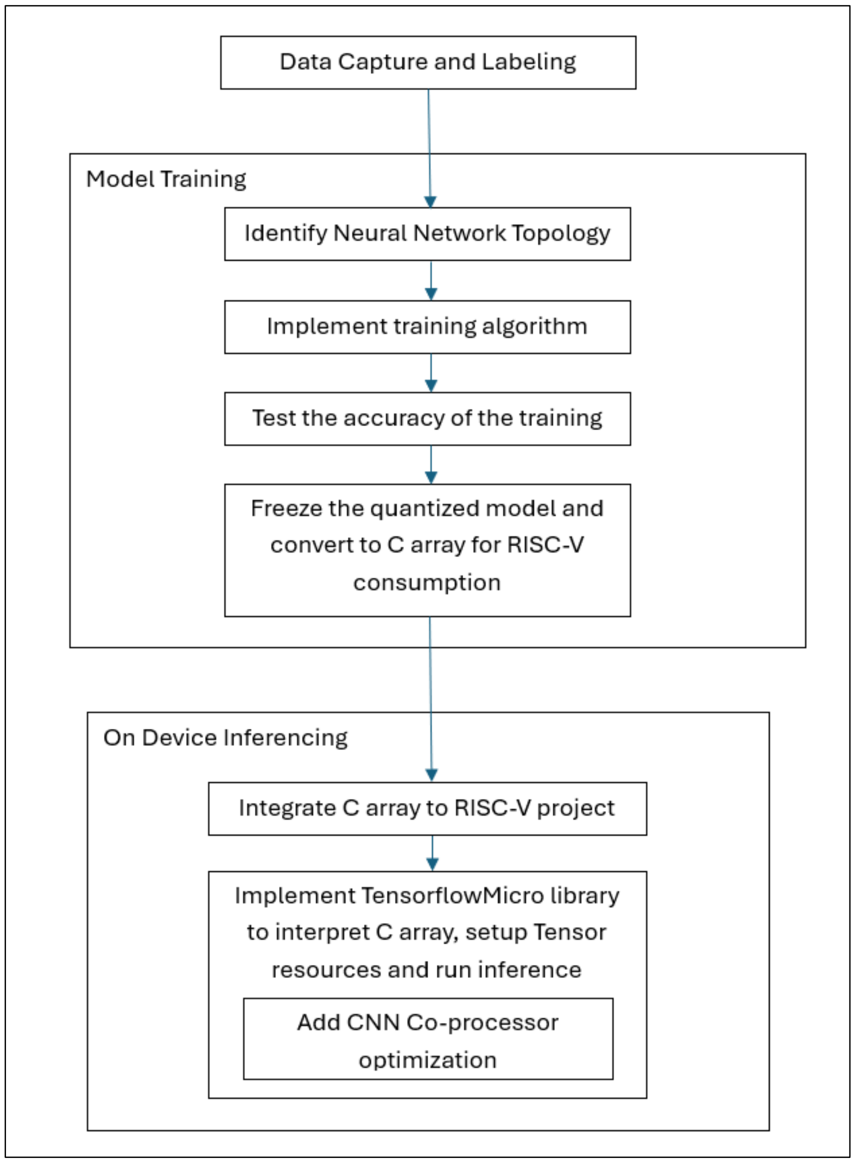


Figure A.1. Predictive Maintenance Machine Learning Overview

A.2. Data Capture and Labeling

The Automate user interface offers essential motor control IP to streamline the data capturing process, as detailed in the Motor Control and PDM Data Collector section. The data format and methodology are further explained in the accompanying whitepaper. Once the motor data is captured, it is categorized into *good* and *bad* data, which are then labeled and stored in folders named *0* and *1* respectively, with *0* indicating good motor data and *1* indicating bad motor data. Note that for this example, the training is performed with data set collected with 800 rpm. Inference is performed with test data ranging from 800 to 1500 rpm.

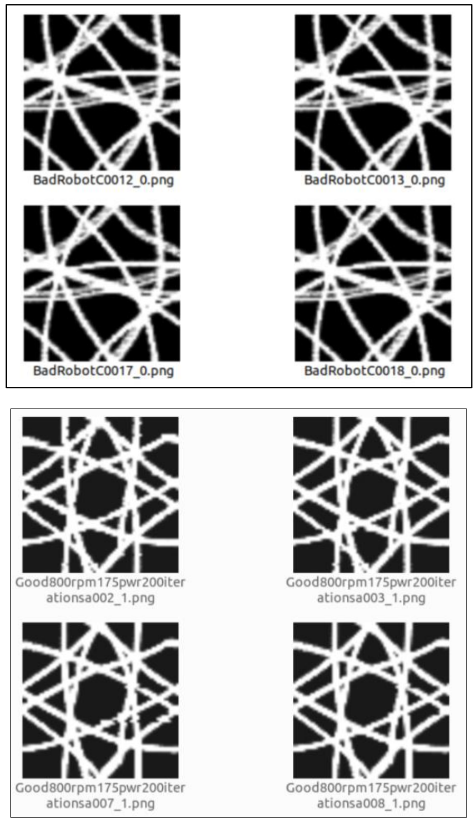


Figure A.2. Data Format Labeling

A.3. Model Training

This section describes the training process outlined in Figure A.1

A.3.1. Training Code Structure

Download the Lattice predictive maintenance demo training code. The link to download the code is available in the Lattice Automate page and the directory structure is shown in Figure A.3. The Identify Neural Network Architecture (Informational) and Implement Training Algorithm sections describe the network topology and background for tuning purposes. The readers need not fully comprehend the details in Identify Neural Network Architecture (Informational) and Implement Training Algorithm. They can proceed with the model training with the details in the Training Framework section that describes a tool to facilitate the process.

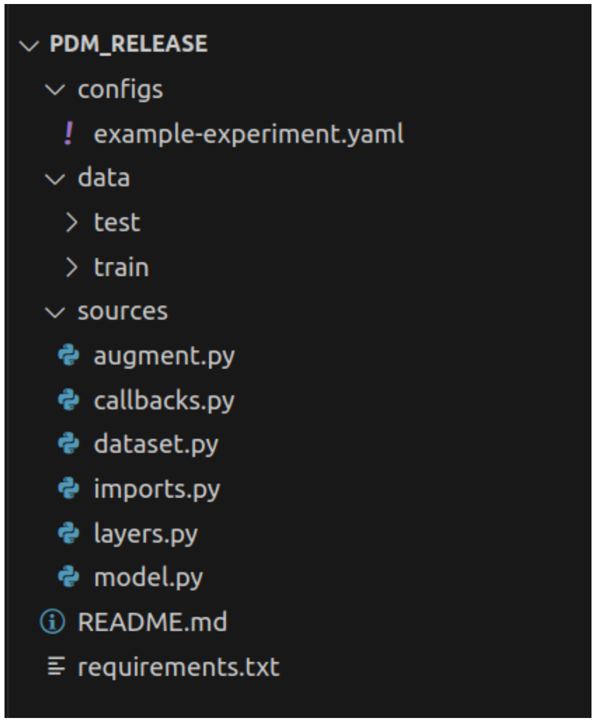


Figure A.3. Training Code Directory Structure

A.3.2. Identify Neural Network Architecture (Informational)

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

Table A.1. Predictive Maintenance Training Network Topology

<table>
<tr><th colspan="3">Input Gray Scale Image (64 x 64 x 1)</th></tr>
<tr><td rowspan="4">Fire1</td><td>Conv3x3 – 8</td><td rowspan="27">Conv3x3 – # where:
Conv3x3 – 3 x 3 Convolution filter Kernel size
- The number of filters
For example, Conv3x3 - 8 = 8 3 x 3 convolution filter

Batchnorm: Batch Normalization
FC – # where:
FC – Fully connected layer
- The number of outputs</td></tr>
<tr><td>Batchnorm</td></tr>
<tr><td>ReLU</td></tr>
<tr><td>Maxpool</td></tr>
<tr><td rowspan="3">Fire2</td><td>Conv3x3 – 8</td></tr>
<tr><td>Batchnorm</td></tr>
<tr><td>ReLU</td></tr>
<tr><td rowspan="4">Fire3</td><td>Conv3x3 – 16</td></tr>
<tr><td>Batchnorm</td></tr>
<tr><td>ReLU</td></tr>
<tr><td>Maxpool</td></tr>
<tr><td rowspan="3">Fire4</td><td>Conv3x3 – 16</td></tr>
<tr><td>Batchnorm</td></tr>
<tr><td>ReLU</td></tr>
<tr><td rowspan="4">Fire5</td><td>Conv3x3 – 16</td></tr>
<tr><td>Batchnorm</td></tr>
<tr><td>ReLU</td></tr>
<tr><td>Maxpool</td></tr>
<tr><td rowspan="3">Fire6</td><td>Conv3x3 – 22</td></tr>
<tr><td>Batchnorm</td></tr>
<tr><td>ReLU</td></tr>
<tr><td rowspan="4">Fire7</td><td>Conv3x3 – 24</td></tr>
<tr><td>Batchnorm</td></tr>
<tr><td>ReLU</td></tr>
<tr><td>Maxpool</td></tr>
<tr><td>Dropout</td><td>Dropout - 0.80</td></tr>
<tr><td>logit</td><td>FC – (3)</td></tr>
</table>

In Table A.1, the 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 the 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 2x2) 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 you know 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, you 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, etc.
  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. The network must 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.3.3. Implement Training Algorithm

The layers described in the previous section are implemented in the code snippet below.

```
def make_resnet_model(input_shape, num_classes, name=None):
    print(input_shape)
    inputs = tf.keras.layers.Input(shape=input_shape[1:], name="input")

    # Fire 1
    x = Conv2D(filters=8, kernel_size=3, strides=1, padding="SAME", use_bias=False, 
name="fire1_conv")(inputs)
    x = BatchNormalization(fused=True, name="fire1_bn")(x)
    x = Activation(activation="relu", name="fire1_relu")(x)
    x = MaxPooling2D(name="fire1_mp")(x)

    # Fire 2
    x = Conv2D(filters=8, kernel_size=3, strides=1, padding="SAME", use_bias=False, 
name="fire2_conv")(x)
    x = BatchNormalization(fused=True, name="fire2_bn")(x)
    x = Activation(activation="relu", name="fire2_relu")(x)

    # Fire 3
    x = Conv2D(filters=16, kernel_size=3, strides=1, padding="SAME", use_bias=False, 
name="fire3_conv")(x)
    x = BatchNormalization(fused=True, name="fire3_bn")(x)
    x = Activation(activation="relu", name="fire3_relu")(x)
    x = MaxPooling2D(name="fire3_mp")(x)

    # Fire 4
    x = Conv2D(filters=16, kernel_size=3, strides=1, padding="SAME", use_bias=False, 
name="fire4_conv")(x)
    x = BatchNormalization(fused=True, name="fire4_bn")(x)
    x = Activation(activation="relu", name="fire4_relu")(x)

    # Fire 5
    x = Conv2D(filters=16, kernel_size=3, strides=1, padding="SAME", use_bias=False, 
name="fire5_conv")(x)
    x = BatchNormalization(fused=True, name="fire5_bn")(x)
    x = Activation(activation="relu", name="fire5_relu")(x)
    x = MaxPooling2D(name="fire5_mp")(x)

    # Fire 6
    x = Conv2D(filters=22, kernel_size=3, strides=1, padding="SAME", use_bias=False, 
name="fire6_conv")(x)
```

```
    x = BatchNormalization(fused=True, name="fire6_bn")(x)
    x = Activation(activation="relu", name="fire6_relu")(x)

    # Fire 7
    x = Conv2D(filters=24, kernel_size=3, strides=1, padding="SAME", use_bias=False,
name="fire7_conv")(x)
    x = BatchNormalization(fused=True, name="fire7_bn")(x)
    x = Activation(activation="relu", name="fire7_relu")(x)
    x = MaxPooling2D(name="fire7_mp")(x)

    x = Dropout(rate=0.8)(x)
    x = Flatten()(x)
    x = Dense(units=num_classes, use_bias=True, activation="linear", name="dense")(x)
    model = tf.keras.Model(inputs=inputs, outputs=[x], name=name)
    return model
```

A.3.4. Training Framework

To streamline the training and testing process, Lattice offers a training tool called Lattice Training Environment (LATTE). This tool is written in Python and is available upon request (refer to Technical Support Assistance). Please refer to the accompanied LATTE document on the installation steps. Once the LATTE package is installed, you have access to the following APIs to execute training, testing and model conversion.

- *latte train* – Trains a model with the specified architecture, producing a model binary as the output.
- *latte test* – Tests the accuracy of the model binary. Typically, 80% of the dataset is used for training, while 20% is reserved for testing.
- *latte convert* – Converts the model to a TensorFlow Lite (tflite) format, suitable for smaller devices such as microcontrollers or RISC-V cores.

After unzipping the code folder, run the commands below:

```
pip install opencv-python  (Note that this is for first time setup only)
latte train configs/example-experiment.yaml sources/imports.py -r
latte test configs/example-experiment.yaml sources/imports.py
latte convert configs/example-experiment.yaml sources/imports.py
```

While LATTE is OS agnostic, the subsequent chapters describe the output of running the above APIs in Linux environment (Ubuntu 20.04). Below printout shows an output report from running “latte train” API running.

```
(fpga) anonymous@anonymous:~/code/MATE/PDM_release$ latte train configs/example-experiment.yaml sources/imports.py -r
[INFO   ] LATTE - Lattice sensAI Neural Networks Training Environment v2.8.0
[INFO   ] Loading and parsing config file 'configs/example-experiment.yaml'
[INFO   ] Loading user code from 'sources/imports.py'
[INFO   ] Info - project: LATTE-Quantization-PDM
[INFO   ] Info - description: Automate Predictive Maintenance with quantization and FPGA Simulator
[WARNING] Sending previous checkpoints to trash
[WARNING] Sending previous best model to trash
[INFO   ] Instantiating model 'PredictiveMaintenanceClassifier'
[None, 64, 64, 3]
1
KerasTensor(type_spec=TensorSpec(shape=(None, 64, 64, 3), dtype=tf.float32, name='input'), name='input', description="created by layer 'input'")
2
KerasTensor(type_spec=TensorSpec(shape=(None, 32, 32, 8), dtype=tf.float32, name=None), name='fire1_mp/MaxPool:0', description="created by layer 'fire1_mp'")
3
KerasTensor(type_spec=TensorSpec(shape=(None, 32, 32, 8), dtype=tf.float32, name=None), name='fire2_relu/Relu:0', description="created by layer 'fire2_relu'")
4
KerasTensor(type_spec=TensorSpec(shape=(None, 16, 16, 16), dtype=tf.float32, name=None), name='fire3_mp/MaxPool:0', description="created by layer 'fire3_mp'")
5
KerasTensor(type_spec=TensorSpec(shape=(None, 16, 16, 16), dtype=tf.float32, name=None), name='fire4_relu/Relu:0', description="created by layer 'fire4_relu'"
6
KerasTensor(type_spec=TensorSpec(shape=(None, 8, 8, 16), dtype=tf.float32, name=None), name='fire5_mp/MaxPool:0', description="created by layer 'fire5_mp'")
7
KerasTensor(type_spec=TensorSpec(shape=(None, 8, 8, 22), dtype=tf.float32, name=None), name='fire6_relu/Relu:0', description="created by layer 'fire6_relu'")
8
KerasTensor(type_spec=TensorSpec(shape=(None, 4, 4, 24), dtype=tf.float32, name=None), name='fire7_mp/MaxPool:0', description="created by layer 'fire7_mp'")
KerasTensor(type_spec=TensorSpec(shape=(None, 3), dtype=tf.float32, name=None), name='dense/BiasAdd:0', description="created by layer 'dense'")
Model: "model"
```

```
[INFO   ] Model number of parameters: 16,067 (15,847 trainable, 220 non-trainable)
[INFO   ] Instantiating and initializing train dataset handler 'PdmDatasetHandler'
[INFO   ] Spliting dataset: keeping 80.0% for training, and 20.0% for evaluation
[INFO   ] Number of epochs: 30
[INFO   ] Training samples: 14,538
[INFO   ] Training batch size: 128
[INFO   ] Training batches: 113
[INFO   ] Evaluation samples: 3,634
[INFO   ] Evaluation batch size: 128
[INFO   ] Evaluation batches: 28
[INFO   ] Best metric is 'accuracy' (higher is better)
[INFO   ] Instantiating optimizer 'Adam'
[INFO   ] Training started on 2024-11-22 16:10:09
[INFO   ] * Epoch  1/30: Train loss: 0.55697 | Eval loss: 0.81389 Eval metrics: accuracy=0.41685 | BEST accuracy 0.41685 | ETA 1m 33s (16:11:46)
[INFO   ] * Epoch  2/30: Train loss: 0.08496 | Eval loss: 0.12720 Eval metrics: accuracy=0.95173 | BEST accuracy 0.95173 (+0.53488) | ETA 59s (16:11:13)
[INFO   ] * Epoch  3/30: Train loss: 0.03526 | Eval loss: 0.00918 Eval metrics: accuracy=0.99693 | BEST accuracy 0.99693 (+0.04520) | ETA 47s (16:11:02)
[INFO   ] * Epoch  4/30: Train loss: 0.02113 | Eval loss: 0.00316 Eval metrics: accuracy=0.99916 | BEST accuracy 0.99916 (+0.00223) | ETA 40s (16:10:56)
[INFO   ]   Epoch  5/30: Train loss: 0.01497 | Eval loss: 0.00540 Eval metrics: accuracy=0.99805 | ETA 36s (16:10:53)
[INFO   ]   Epoch  6/30: Train loss: 0.00964 | Eval loss: 0.00337 Eval metrics: accuracy=0.99860 | ETA 24s (16:10:42)
[INFO   ]   Epoch  7/30: Train loss: 0.00990 | Eval loss: 0.00951 Eval metrics: accuracy=0.99526 | ETA 23s (16:10:42)
[INFO   ] * Epoch  8/30: Train loss: 0.00640 | Eval loss: 0.00102 Eval metrics: accuracy=0.99972 | BEST accuracy 0.99972 (+0.00056) | ETA 22s (16:10:42)
[INFO   ]   Epoch  9/30: Train loss: 0.00442 | Eval loss: 0.00081 Eval metrics: accuracy=0.99972 | ETA 21s (16:10:42)
[INFO   ]   Epoch 10/30: Train loss: 0.00403 | Eval loss: 0.00081 Eval metrics: accuracy=0.99972 | ETA 20s (16:10:42)
[INFO   ] * Epoch 11/30: Train loss: 0.00281 | Eval loss: 0.00039 Eval metrics: accuracy=1.00000 | BEST accuracy 1.00000 (+0.00028) | ETA 19s (16:10:42)
[INFO   ]   Epoch 12/30: Train loss: 0.00225 | Eval loss: 0.00040 Eval metrics: accuracy=0.99972 | ETA 18s (16:10:42)
[INFO   ]   Epoch 13/30: Train loss: 0.00187 | Eval loss: 0.00095 Eval metrics: accuracy=0.99916 | ETA 16s (16:10:42)
[INFO   ]   Epoch 14/30: Train loss: 0.00176 | Eval loss: 0.00108 Eval metrics: accuracy=0.99972 | ETA 16s (16:10:43)
[INFO   ]   Epoch 15/30: Train loss: 0.00098 | Eval loss: 0.00010 Eval metrics: accuracy=1.00000 | ETA 15s (16:10:43)
[INFO   ]   Epoch 16/30: Train loss: 0.00270 | Eval loss: 0.00075 Eval metrics: accuracy=0.99972 | ETA 14s (16:10:43)
[INFO   ]   Epoch 17/30: Train loss: 0.00154 | Eval loss: 0.00059 Eval metrics: accuracy=0.99972 | ETA 13s (16:10:43)
[INFO   ]   Epoch 18/30: Train loss: 0.00197 | Eval loss: 0.00840 Eval metrics: accuracy=0.99526 | ETA 12s (16:10:43)
[INFO   ]   Epoch 19/30: Train loss: 0.00191 | Eval loss: 0.00005 Eval metrics: accuracy=1.00000 | ETA 10s (16:10:43)
[INFO   ]   Epoch 20/30: Train loss: 0.00076 | Eval loss: 0.00009 Eval metrics: accuracy=1.00000 | ETA 9s (16:10:43)
[INFO   ]   Epoch 21/30: Train loss: 0.00068 | Eval loss: 0.00072 Eval metrics: accuracy=0.99972 | ETA 8s (16:10:43)
[INFO   ]   Epoch 22/30: Train loss: 0.00232 | Eval loss: 0.00127 Eval metrics: accuracy=0.99972 | ETA 7s (16:10:43)
[INFO   ]   Epoch 23/30: Train loss: 0.00207 | Eval loss: 0.00204 Eval metrics: accuracy=0.99916 | ETA 7s (16:10:43)
[INFO   ]   Epoch 24/30: Train loss: 0.00202 | Eval loss: 0.00107 Eval metrics: accuracy=1.00000 | ETA 5s (16:10:43)
[INFO   ]   Epoch 25/30: Train loss: 0.00102 | Eval loss: 0.00002 Eval metrics: accuracy=1.00000 | ETA 4s (16:10:43)
[INFO   ]   Epoch 26/30: Train loss: 0.00128 | Eval loss: 0.00125 Eval metrics: accuracy=0.99972 | ETA 4s (16:10:43)
[INFO   ]   Epoch 27/30: Train loss: 0.00115 | Eval loss: 0.00068 Eval metrics: accuracy=0.99972 | ETA 3s (16:10:43)
[INFO   ]   Epoch 28/30: Train loss: 0.00151 | Eval loss: 0.00006 Eval metrics: accuracy=1.00000 | ETA 2s (16:10:43)
[INFO   ]   Epoch 29/30: Train loss: 0.00042 | Eval loss: 0.00002 Eval metrics: accuracy=1.00000 | ETA 1s (16:10:43)
[INFO   ]   Epoch 30/30: Train loss: 0.00025 | Eval loss: 0.00001 Eval metrics: accuracy=1.00000
[INFO   ] BEST accuracy = 1.00000 at epoch 11
[INFO   ] Training finished on 2024-11-22 16:10:44 (duration: 34s)
```

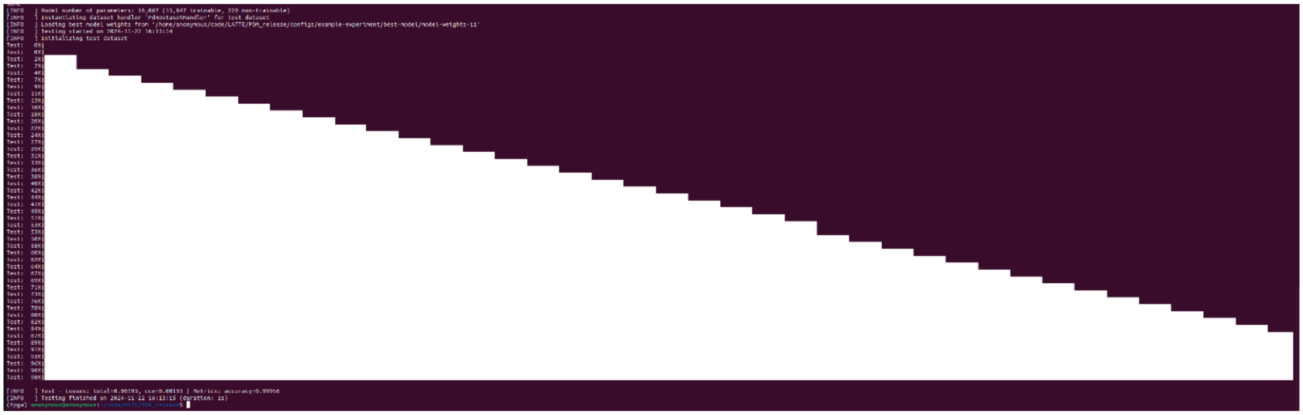
Figure A.4. LATTE Training

A.3.5. Testing the Accuracy of the Trained Model

The LATTE API *latte test* facilitates testing of the trained model using FPGA simulator. The FPGA simulator needs to be installed to simulate the model accuracy running on FPGA. The FPGA simulator is included in the LATTE release package.

The trained model provided in Automate 4.0 achieves the test accuracy of 99.956% as shown in Figure A.5..

```
(fpga) anonymous@anonymous:~/code/MATE/PDM_release$ latte test configs/example-experiment.yaml sources/imports.py
[INFO    ] LATTE - Lattice sensAI Neural Networks Training Environment v2.8.0
[INFO    ] Loading and parsing config file 'configs/example-experiment.yaml'
[INFO    ] Loading user code from 'sources/imports.py'
[INFO    ] Info - project: LATTE-Quantization-PDM
[INFO    ] Info - description: Automate Predictive Maintenance with quantization and FPGA Simulator
[INFO    ] Instantiating model 'PredictiveMaintenanceClassifier'
```



```
[INFO    ] Test - Losses: total=0.00193, cce=0.00193 | Metrics: accuracy=0.99956
[INFO    ] Testing finished on 2024-11-22 16:13:15 (duration: 1s)
(fpga) anonymous@anonymous:~/code/MATE/PDM_release$
```

Figure A.5. LATTE Testing

A.3.6. Converting the Trained Model

The LATTE API *latte convert* facilitates conversion of the model to deploy on FPGA. The output of this command is a tflite model file.

```
(fpga) anonymous@anonymous:~/code/MATE/PDM_release$ latte convert configs/example-experiment.yaml sources/imports.py
[INFO    ] LATTE - Lattice sensAI Neural Networks Training Environment v2.8.0
[INFO    ] Loading and parsing config file 'configs/example-experiment.yaml'
[INFO    ] Loading user code from 'sources/imports.py'
[INFO    ] Info - project: LATTE-Quantization-PDM
[INFO    ] Info - description: Automate Predictive Maintenance with quantization and FPGA Simulator
[INFO    ] Instantiating model 'PredictiveMaintenanceClassifier'
```

```
INFO    ] Model number of parameters: 16,067 (15,847 trainable, 220 non-trainable)
INFO    ] Loading best model weights from '/home/anonymous/code/LATTE/PDM_release/configs/example-experiment/best-model/model-weights-11'
INFO    ] Conversion started on 2024-11-22 16:14:37
INFO    ] Running converter 'tflite'
WARNING] Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolut
on_op while saving (showing 5 of 7). These functions will not be directly callable after loading.
INFO    ] Model converted to /home/anonymous/code/LATTE/PDM_release/configs/example-experiment/convert/model-tflite-best.tflite
INFO    ] Conversion finished on 2024-11-22 16:14:40 (duration: 3s)
fpga) anonymous@anonymous:~/code/MATE/PDM_release$
```

Figure A.6. LATTE Conversion

The tflite model is then translated into a C-Array using the xxd tool. This step is necessary for the model to be recognized by the FPGA RISC-V. The quickest way to run xxd in Windows environment is by installing Git Bash terminal and run the xxd command with it.

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

```
tloo@LPGL109105 MINGW64 ~
$ xxd --version
xxd 2022-01-14 by Juergen Weigert et al.

tloo@LPGL109105 MINGW64 ~
$ cd Downloads/

tloo@LPGL109105 MINGW64 ~/Downloads
$ xxd -i new_model_quant.tflite > mymodel.cc

tloo@LPGL109105 MINGW64 ~/Downloads
$ cat mymodel.cc
unsigned char new_model_quant_tflite[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
  0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10, 0x00, 0x0c, 0x00, 0x00, 0x00,
  0x08, 0x00, 0x04, 0x00, 0x14, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00,
  0x80, 0x00, 0x00, 0x00, 0xd8, 0x00, 0x00, 0x00, 0x34, 0x41, 0x00, 0x00,
  0x44, 0x41, 0x00, 0x00, 0x84, 0x5f, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00,
  0x01, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x26, 0xbe, 0xff, 0xff,
  0x0c, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00, 0x38, 0x00, 0x00, 0x00,
```

Figure A.7.Model to C Array Conversion

A.4. On Device Inferencing

Th inference is executed on the main system RISC-V core. The RISC-V core requires a library to understand the converted trained model, which is the TensorFlowMicro library described in A.4.1.

A.4.1. Implementing the TensorflowMicro Library

The Automate 4.0 main project offers a reference for integrating the TensorFlow Micro library into a RISC-V project. This library is based on an open-source implementation, which can be accessed in the Tensorflow Lite Micro Github page.

The algorithm divided into two parts: setup and compute. The setup() function handles target initialization, tensor arena size allocation, model retrieval (the c array), mutable operation resolver allocation, interpreter addition, and tensor memory allocation. The compute function then processes the input data (motor data) and runs the invoke function, which returns the inference outcome. The inference output provides a confidence level number for both good and bad categories for each input data. The category with the higher confidence value determines the inference result.

Table A.1.Example of Inference Outcomes

Input (Motor data)	Inference Output (Good)	Inference Output (Bad)	Summary
a	53	-12	Good condition
b	7	23	Maintenance required

A.4.2. CNN Co-processor Optimization

The CNN co-processor IP, as described in the CNN Co-Processor Unit (CCU) section, is required to enhance convolution operations. To utilize the CNN co-processor effectively, you need to use the modified conv.h file available in the RISC-V Propel project. With the optimization provided by the CNN coprocessor, the inference process time is significantly reduced.

Appendix B. Setting Up the Wireshark Tool

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

To set up the wireshark tool, perform the following:

1. Open the Wireshark tool and select the network (Ethernet).
2. Click on the Ethernet network.
3. Click the **Run** ( ) button.
4. Check the UDP message use port filter *udp.port == 1486* on the top bar.

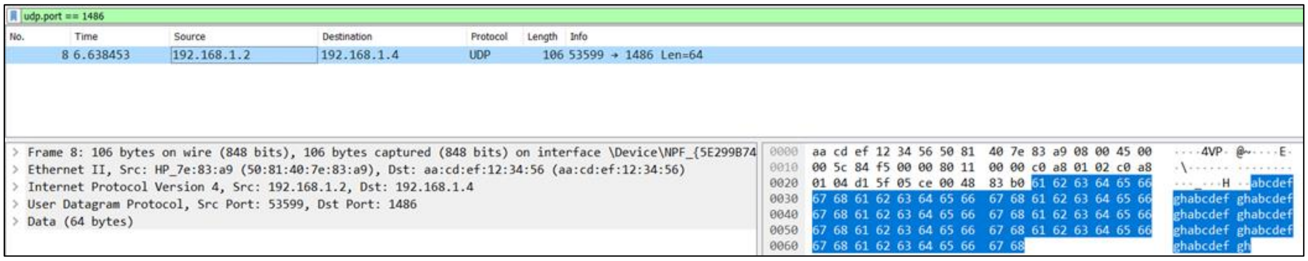


Figure B.1. Wireshark Tool – Write udp.port == 1486

5. Check both the source and destination IP.

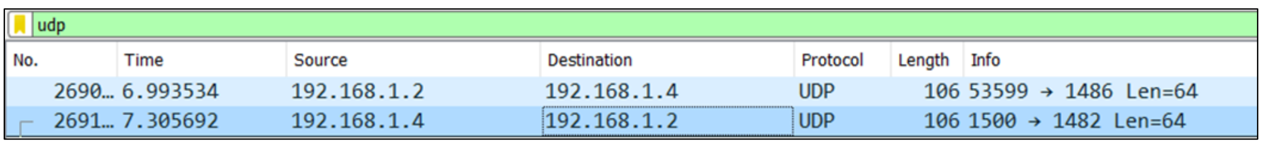


Figure B.2. Source and Destination UDP Packet

6. Click on the UDP packet.

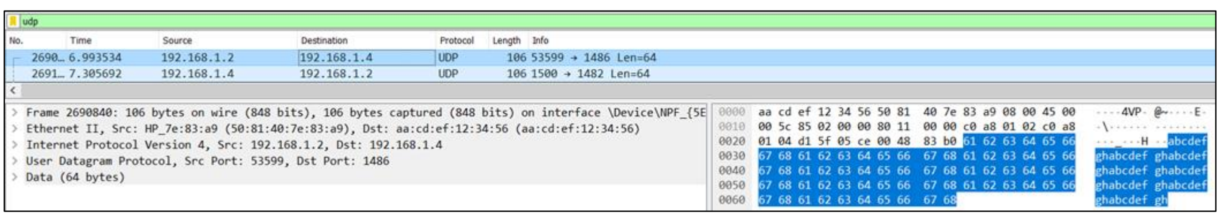


Figure B.3. Wireshark tool – First UDP Packet

Appendix C. Automate Stack 4.0 Bit and Binary Generation

C.1. Installing the Propel SDK 2024.1

For steps on installing the Propel 2024.1, see the Official Documentation and Training page.

C.2. Installing the Propel Patch 2024.1

To install the Propel Patch, perform the following:

1. Double-click on the application to install the patch.

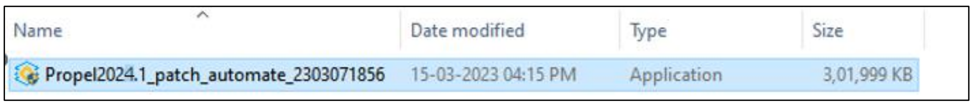


Figure C.1. Propel Patch Application

2. Click **Next** on the succeeding windows.

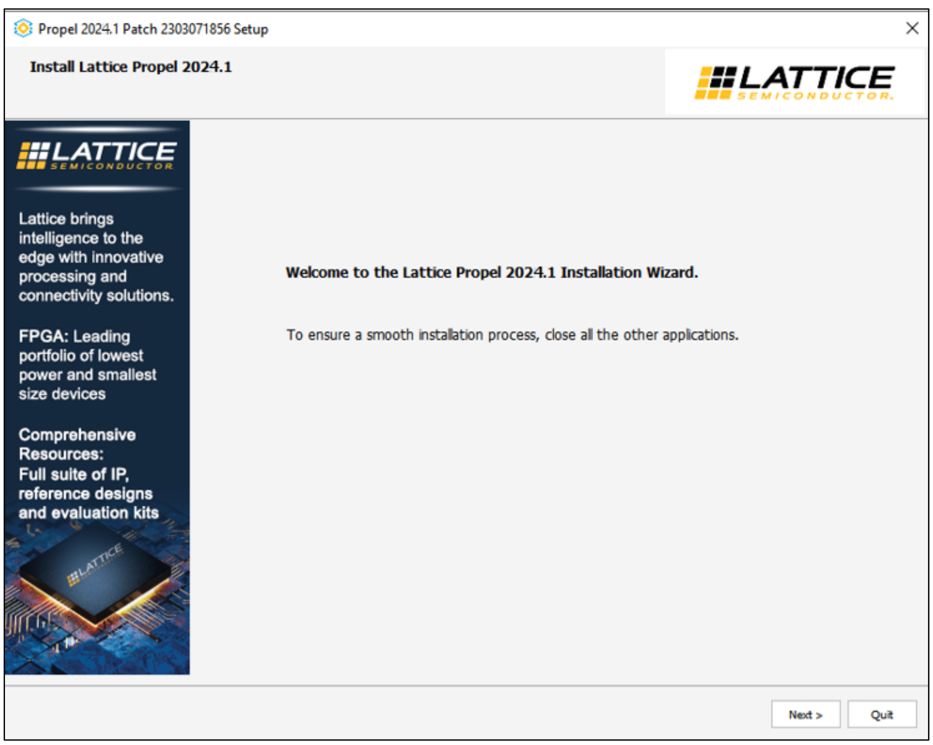


Figure C.2. Propel Patch Setup Window – Install Lattice Propel

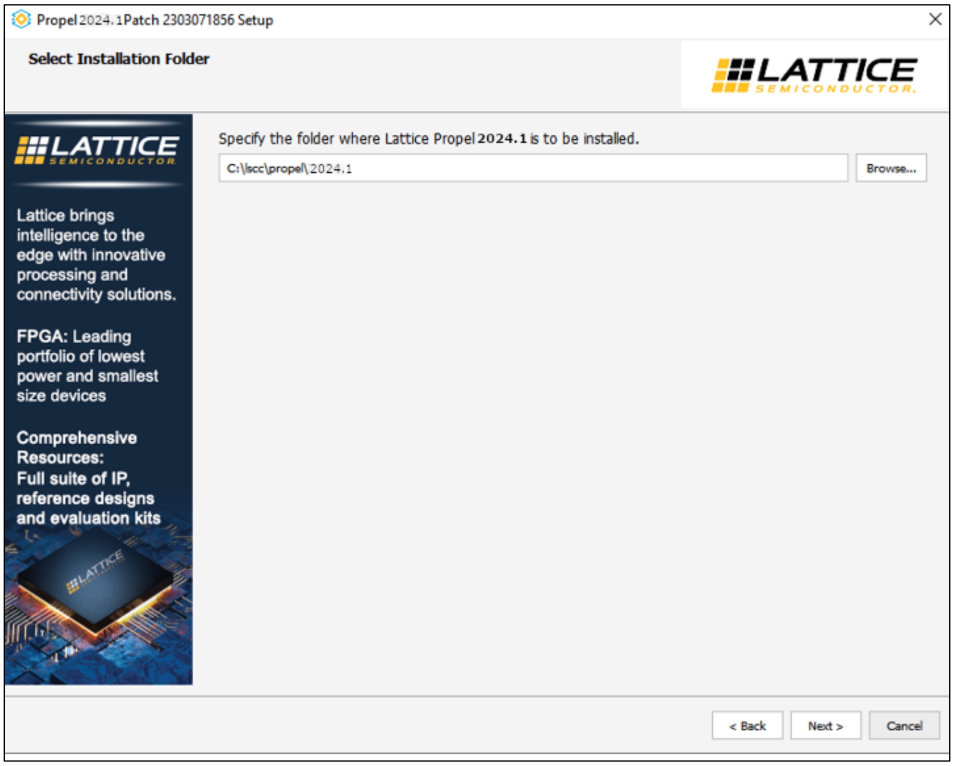


Figure C.3. Propel Patch Setup Window – Select Installation Folder

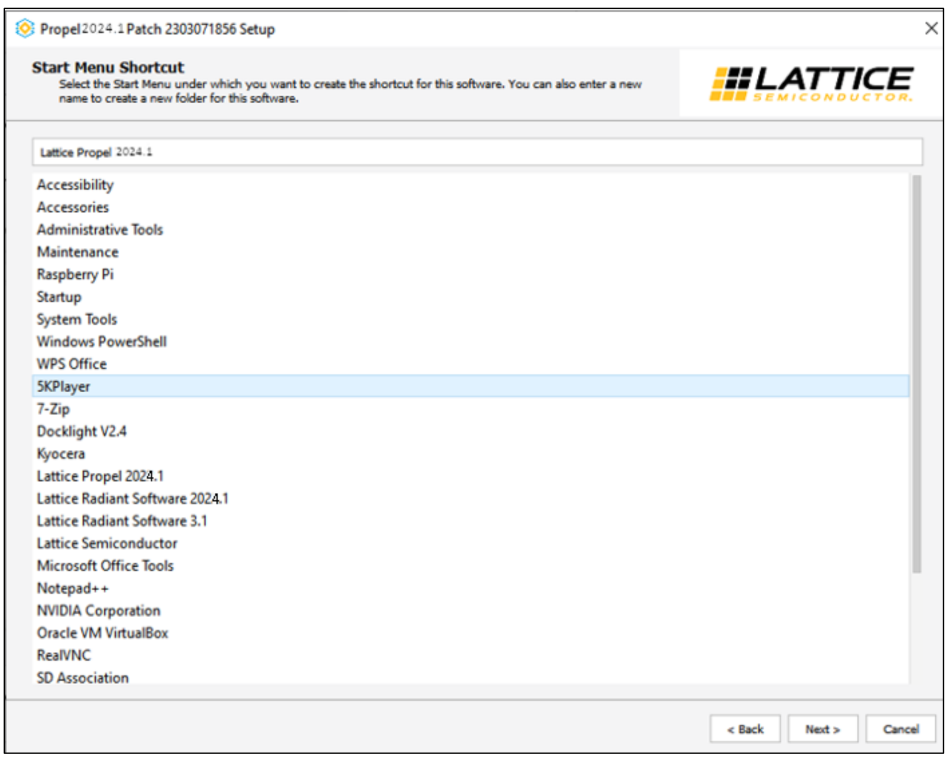


Figure C.4. Propel Patch Setup Window – Start Menu Shortcut

3. Wait for the installation process to 100%.

Figure C.5. Installation Process

4. Click **Finish**.

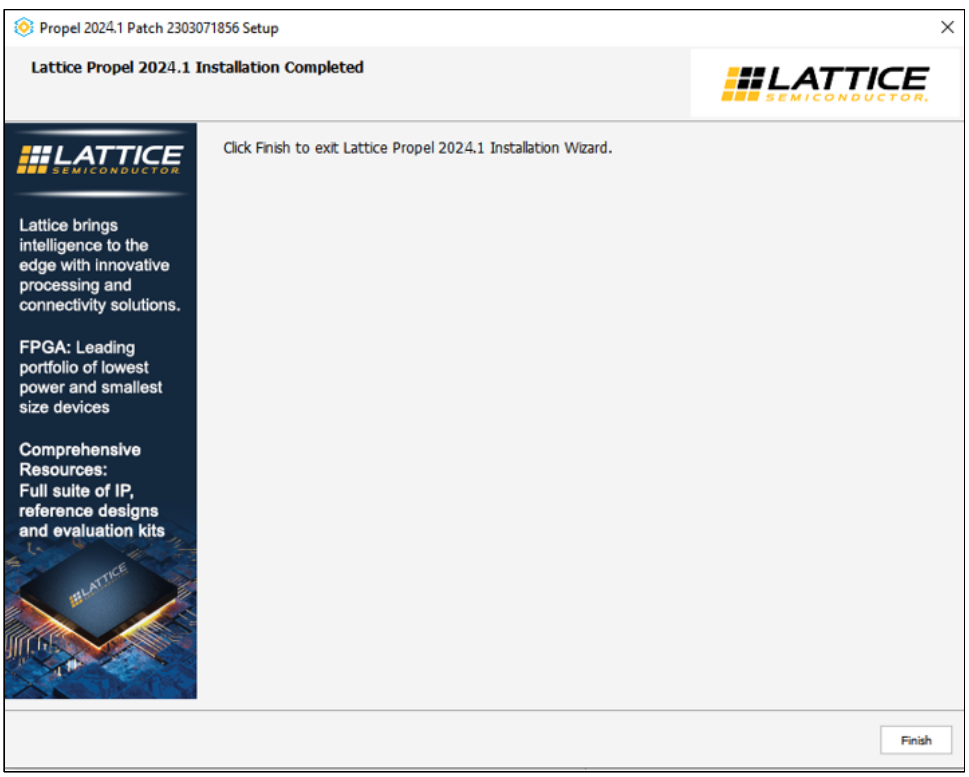


Figure C.6. Installation Completed

C.3. Generating the Binary in the Main System

C.3.1. Primary Main System

To generate the binary in the primary main system, perform the following:

1. Double-click **Lattice Propel SDK 2023.2** to open the dialogue box as shown in Figure C.7.

Figure C.7. Propel 2024.1 Application

2. To select the workspace, browse to the template location or where your project is located.
3. Select **\Main_System\Primary_MainSystem** by clicking on **Browse**. Click **Launch** to launch the workspace.

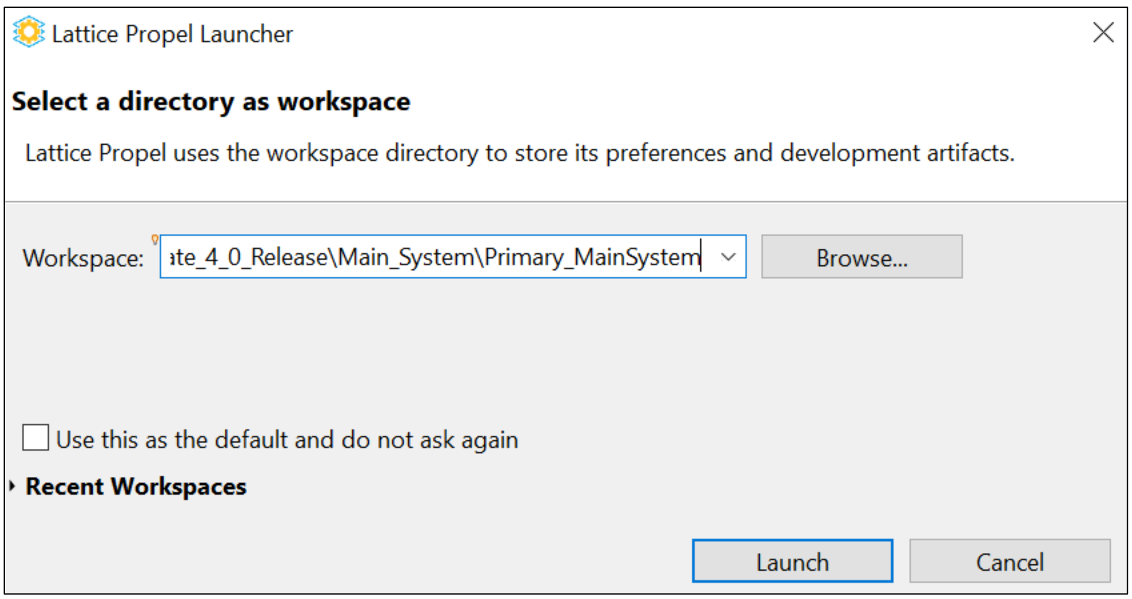


Figure C.8. Select Directory

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

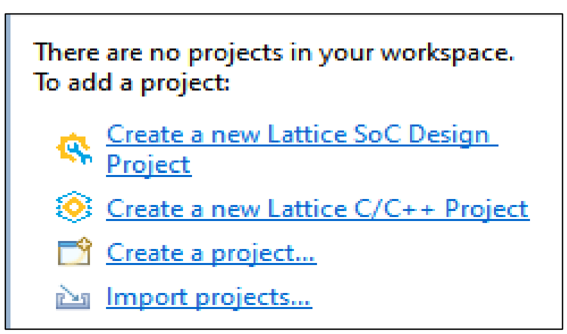


Figure C.9. Import Project

5. Select the existing project in the workspace from the general list and click **Next**.

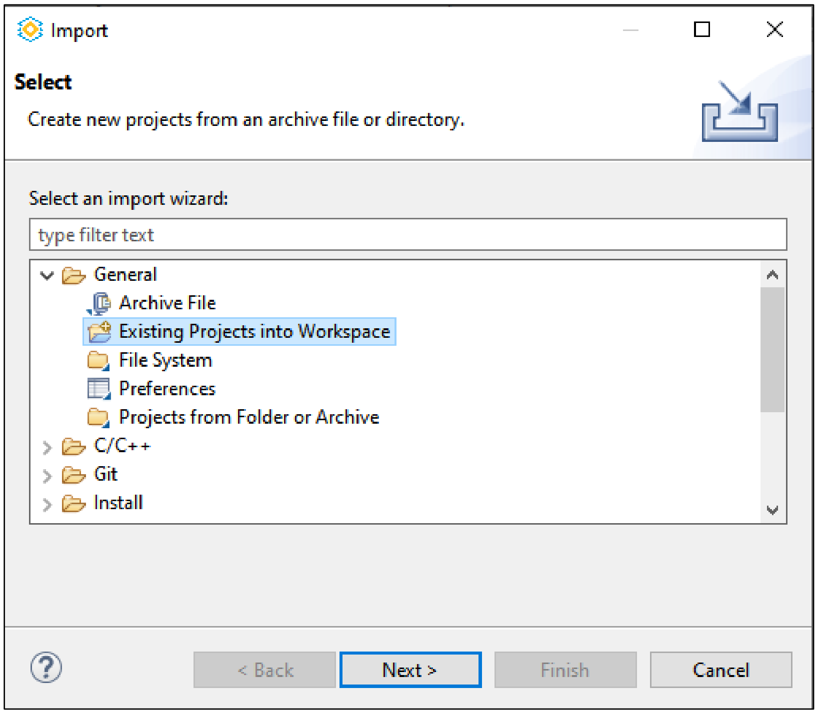


Figure C.10. Existing Project into Workspace

6. Select the root directory and browse template location.
7. Select the project as shown in Figure C.11: **\Main_System\Primary_MainSystem**.
8. Click **Finish**.

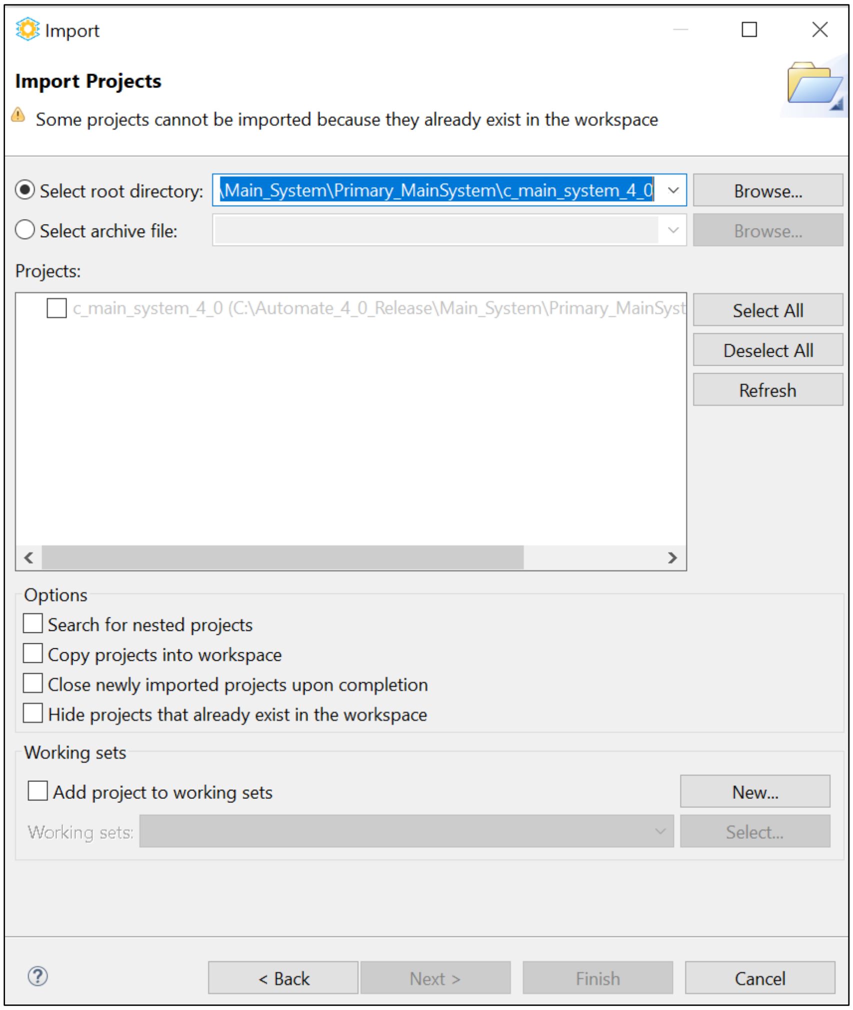


Figure C.11. Import Project

9. Right-click on the firmware project folder *c_main_system_4_0* and select **Properties**.

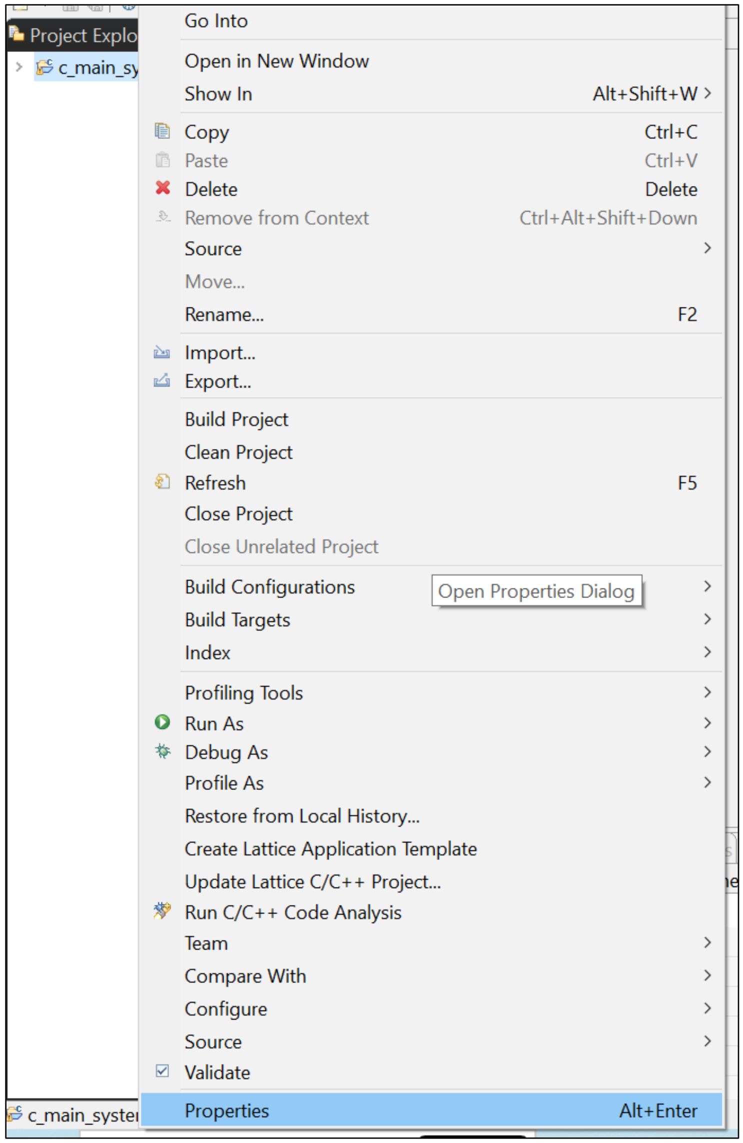


Figure C.12. Properties

10. Drop-down the c/c++ build and select **Settings**. Click **Manage configuration**.

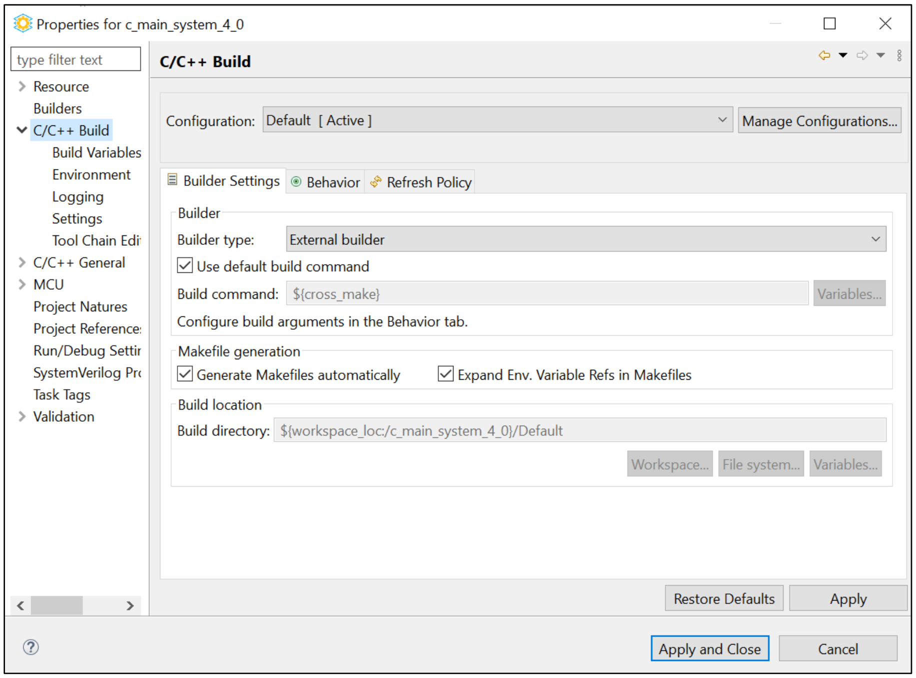


Figure C.13. C/C++ Build Settings

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

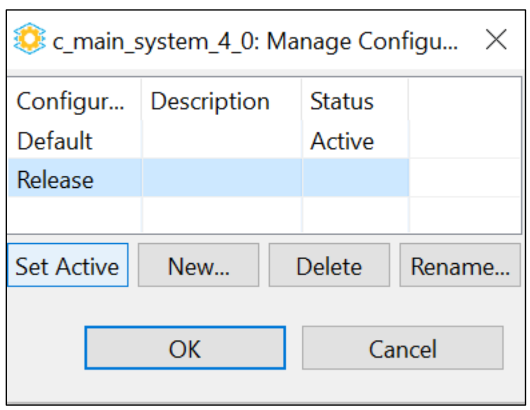


Figure C.14. Manage Configuration – Release: Set Active

12. Click **Apply and Close**.

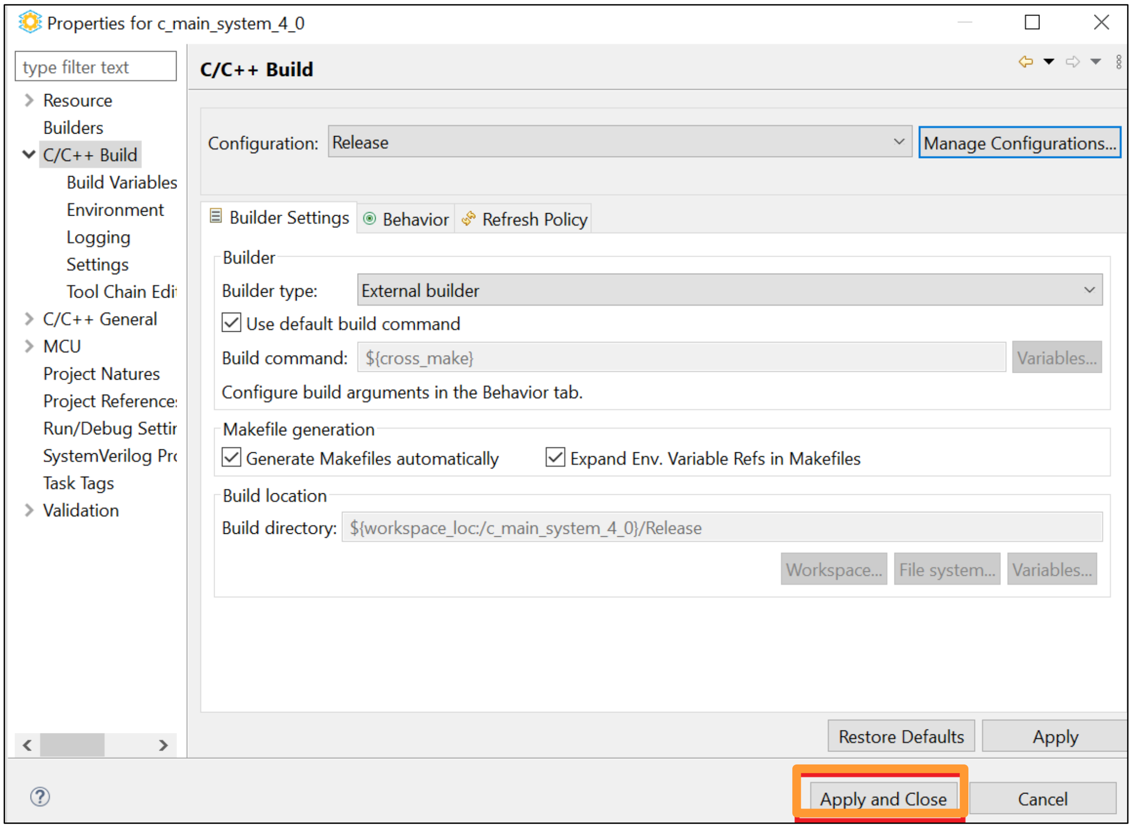


Figure C.15. Manage Configuration: Apply and Close

13. Right-click on the firmware project folder *c_main_system_4_0* and select the option as shown in Figure C.16. to build the project.

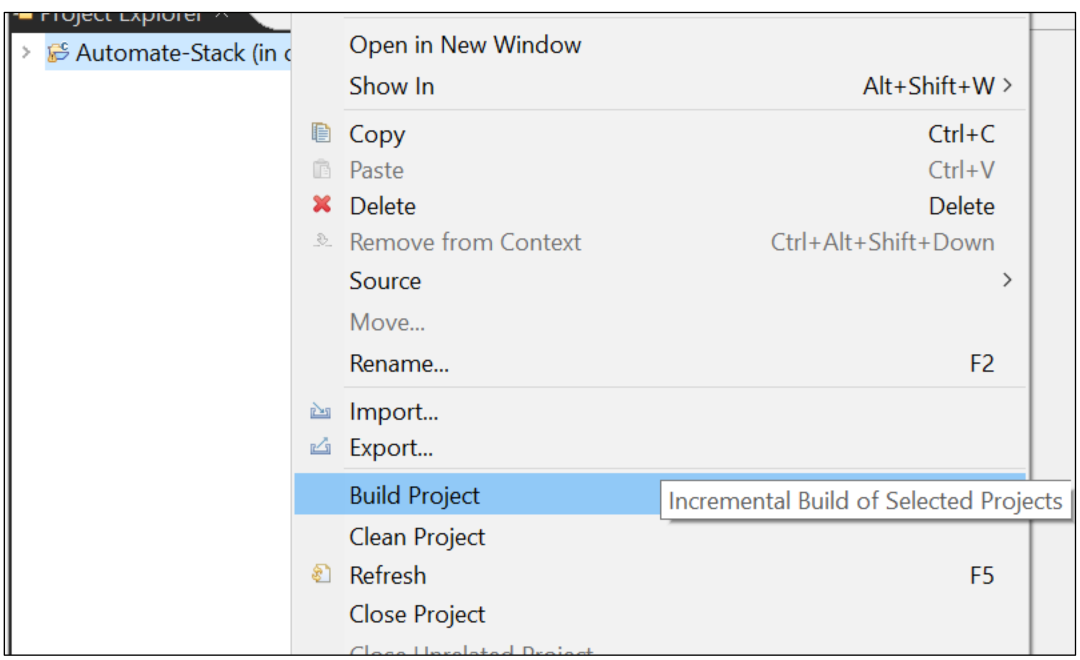


Figure C.16. Build Project

14. Wait for the process to complete to 100%. After completion, the message shown in Figure C.17 appears on the console.

```
Finished building: c_main_system_4_0cnn.lst

srec_cat.exe "@..\crc_add.txt" && srec_cat.exe "@..\nocrc_add.txt"

10:10:44 Build Finished. 0 errors, 0 warnings. (took 1m:22s.948ms)
```

Figure C.17. Completing Process

15. To locate the binary file to below path: **\Main_System\Primary_MainSystem\c_main_system_4_0_cnn\Release**.

C.3.2. Golden Main System

To generate the binary in the golden main system, perform the following:

1. Double-click **Lattice Propel SDK 2024.1** to open the dialogue box as shown in below fig.

Figure C.18. Propel 2024.1 application

2. To select the workspace, browse to the template location or where your project is located.
3. Select **MainSystem\Golden_MainSystem** by clicking on the **Browse** option as shown below. Click **Launch** to launch the workspace.

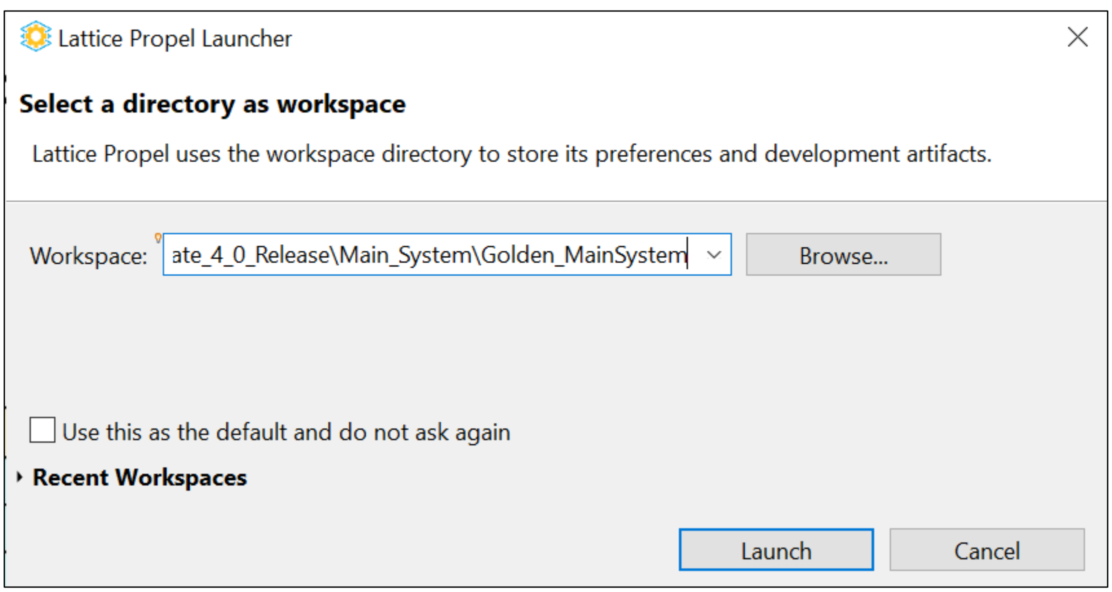


Figure C.19. Select Directory

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

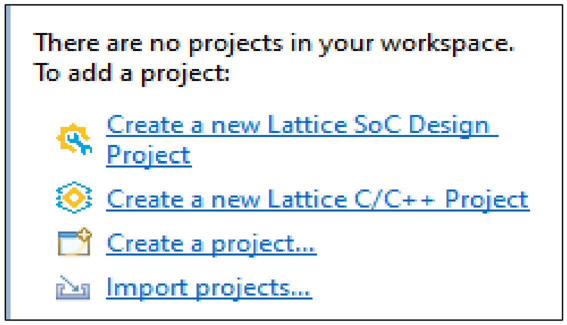


Figure C.20. Import Project

5. Select **Existing Project in Workspace** from the General list and click **Next** as shown below.

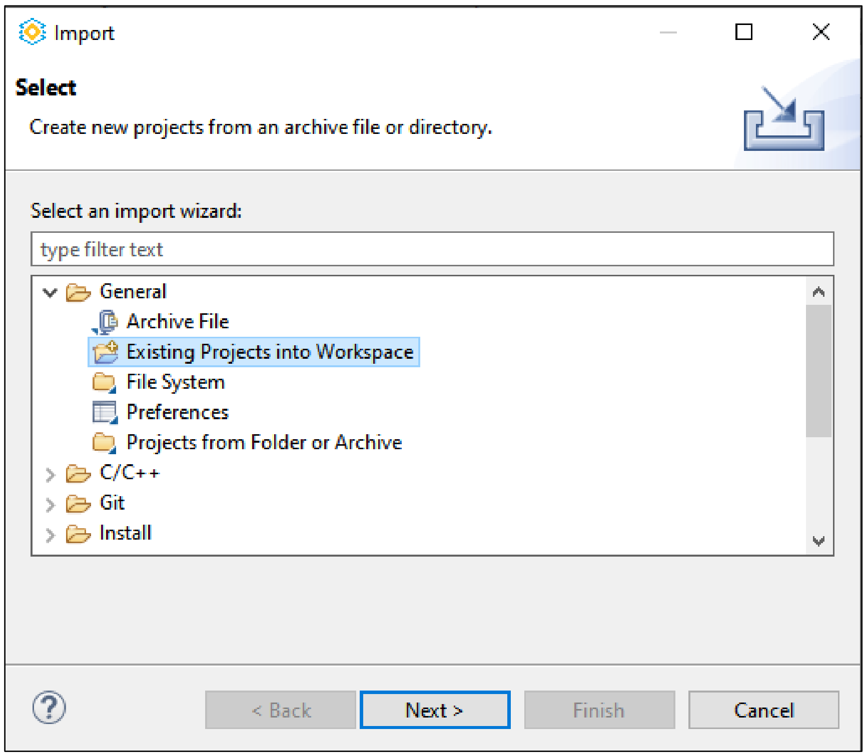


Figure C.21. Existing Project into Workspace

6. Select the root directory and browse template location.
7. Select the project as shown in Figure C.22: **\MainSystem\Golden_MainSystem**.
8. Click **Finish**.

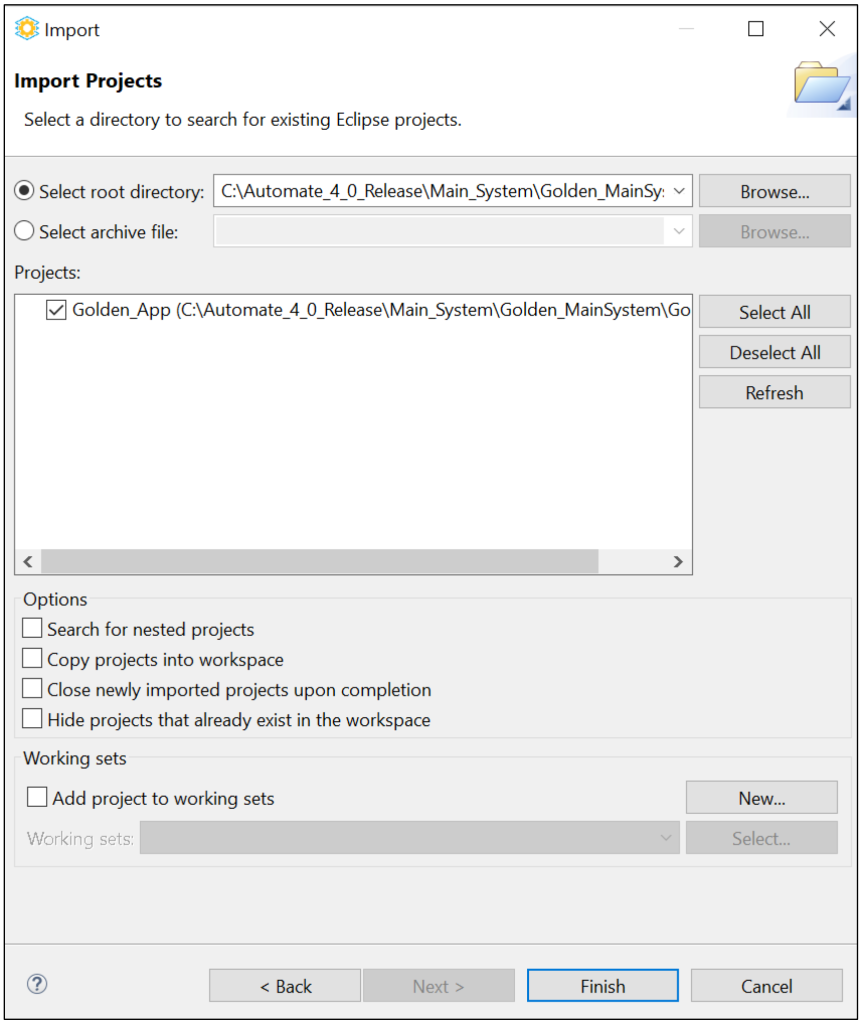


Figure C.22. Import Project

9. Right click on the firmware project folder **Golden_App** and select the option as shown in Figure C.23 to build the project.

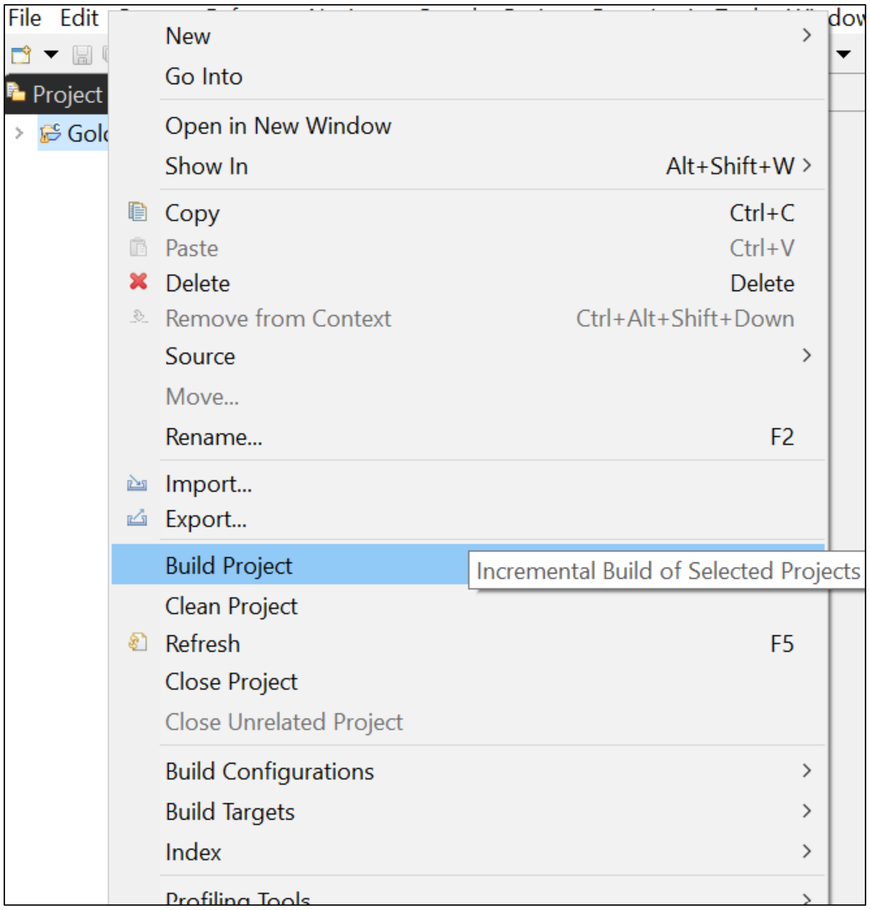


Figure C.23. Build Project

10. Wait for the process to complete to 100%. After completion, the message shown below appears on the console.

```
Finished building: Golden_App.lst

srec_cat.exe "@..\crc_add.txt" && srec_cat.exe "@..\nocrc_add.txt"

10:47:00 Build Failed. 1 errors, 0 warnings. (took 8m:11s.764ms)
```

Figure C.24. Completing Process

11. To locate the binary file to below path: **\MainSystem\Golden_MainSystem\Golden_App\Release**.

C.3.3. Node System

To generate the binary in the node system, perform the following:

1. Double-click **Lattice Propel SDK 2024.1** to open the dialogue box.

Figure C.25. Propel Application

2. To select the workspace, browse to the template location **\NodeSystem** by clicking on the **Browse** option as shown below. Click **Launch** to launch the workspace.

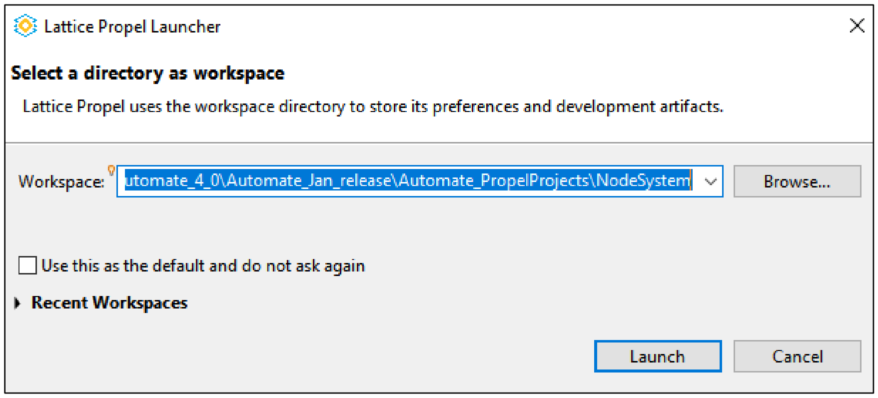


Figure C.26. Select Directory

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

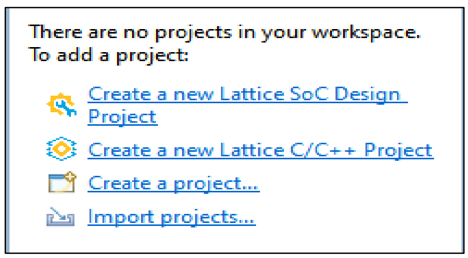


Figure C.27. Import Project

4. Select **Existing Project in Workspace** from the General list and click **Next**.

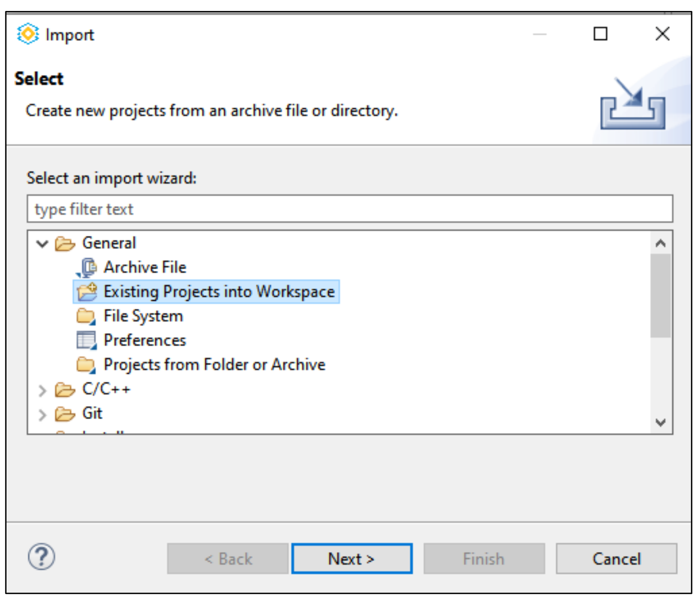


Figure C.28. Existing Project into Workspace

5. Select root directory and browse template location.

6. Select project as shown in below: **\NodeSystem**.
7. Click **Finish**.

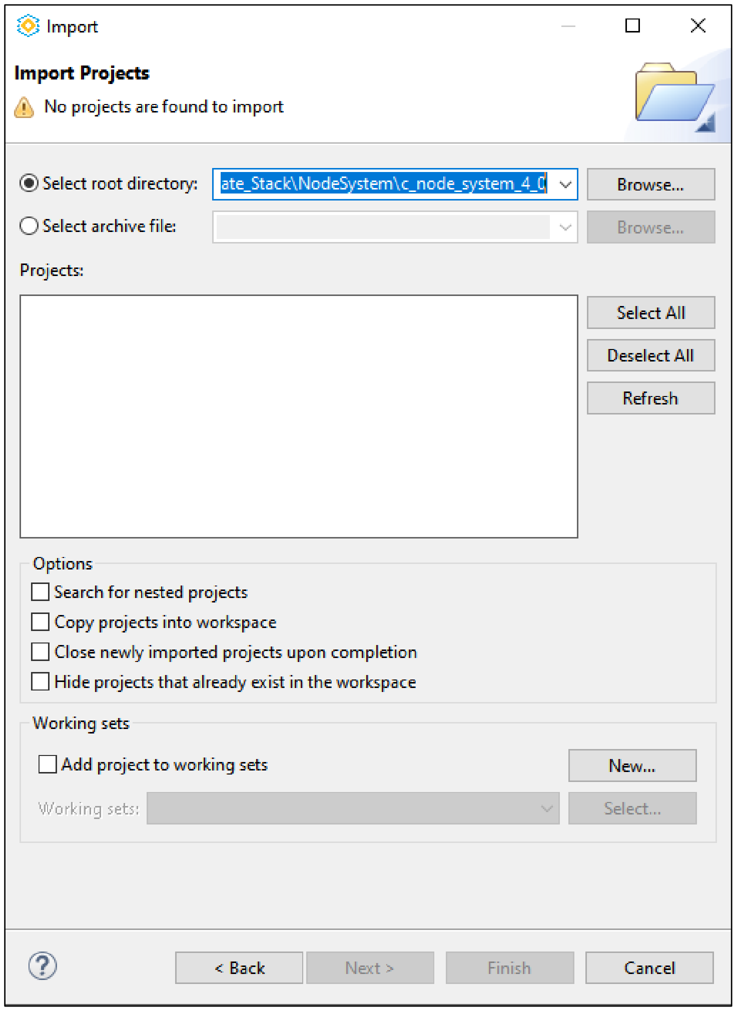


Figure C.29. Select Project

8. Right-click on the firmware project folder *c_node_system_4_0* and select the option as shown in Figure C.30 to clean the project before building.

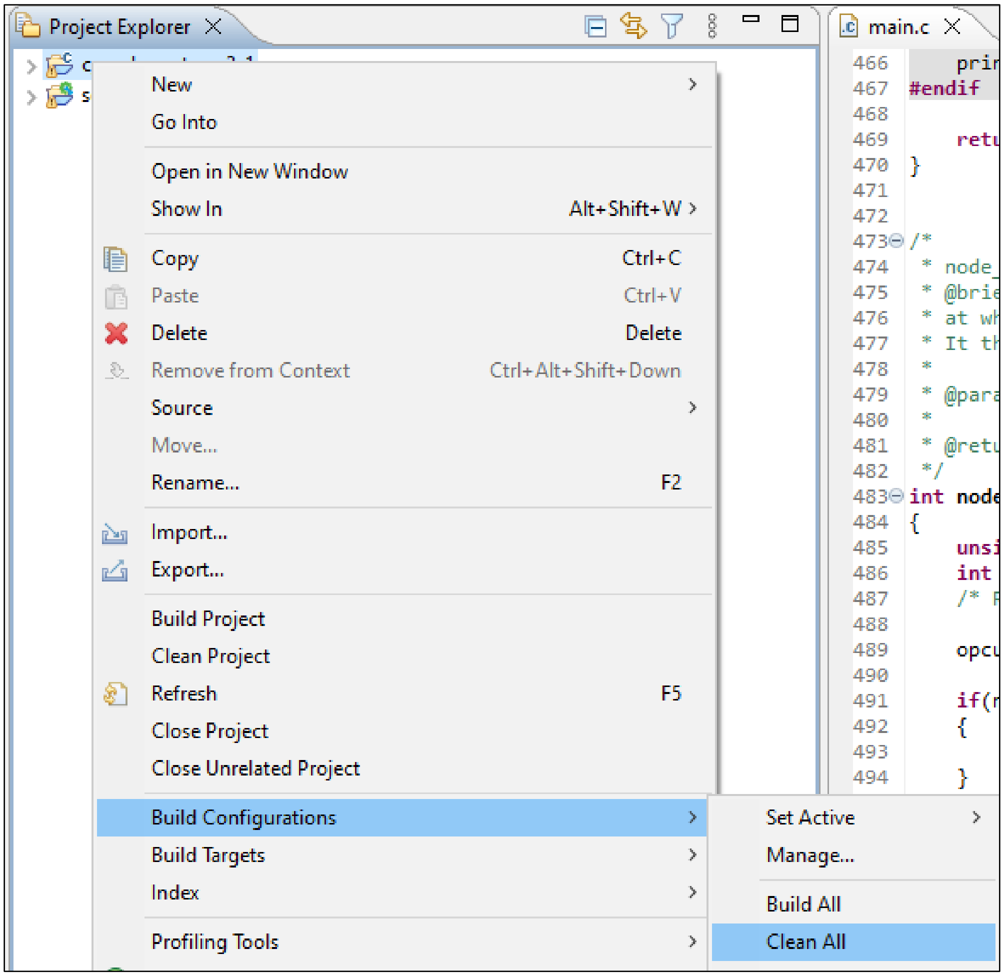


Figure C.30. Clean All

9. After selecting the option as shown in Figure C.30, observe the console and wait for the process to complete to 100%. After completion, the message shown below appears on the console.

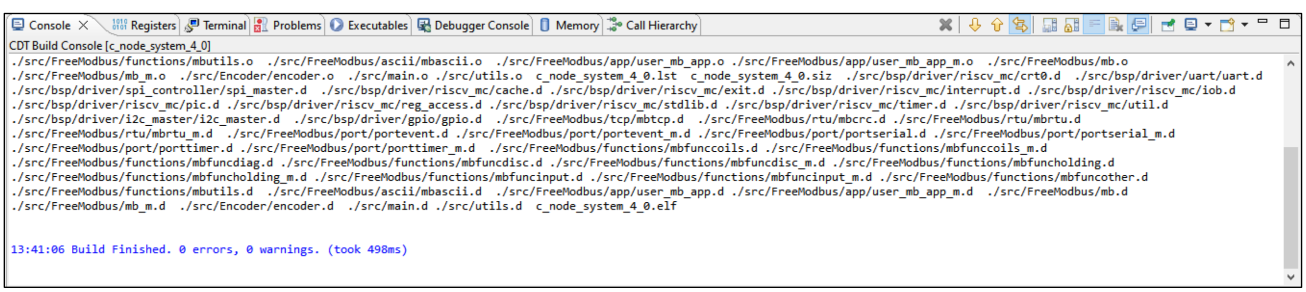


Figure C.31. Console

10. After cleaning, right-click on the *c_node_system_4_0* and select the option as shown in Figure C.32 to build the project.

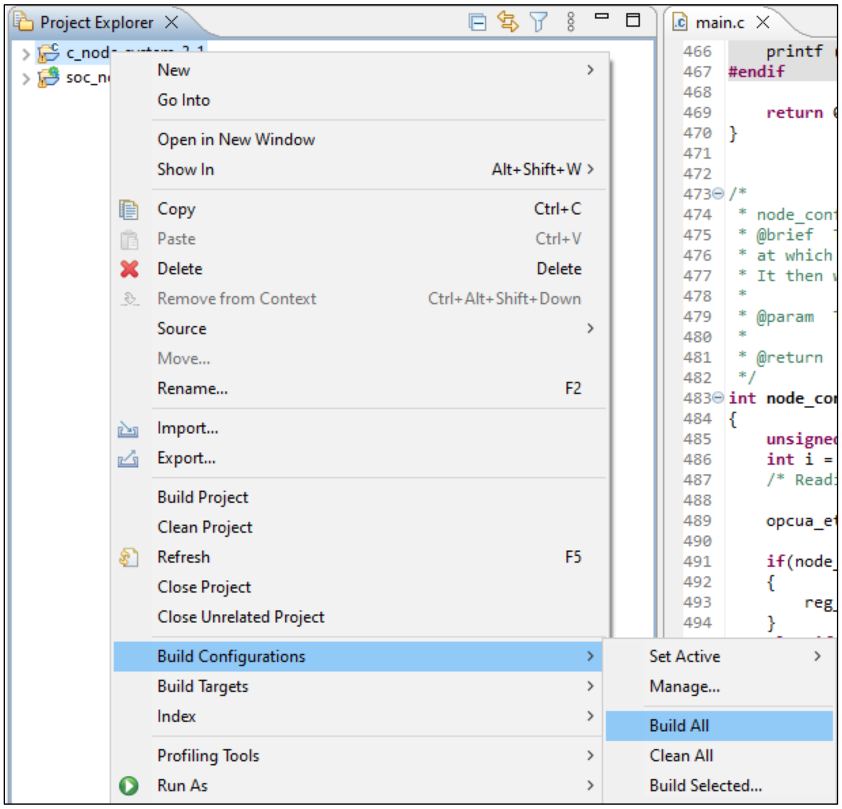


Figure C.32. Build All

11. Wait for the process to complete to 100%. After completion, the message below appears on the console.

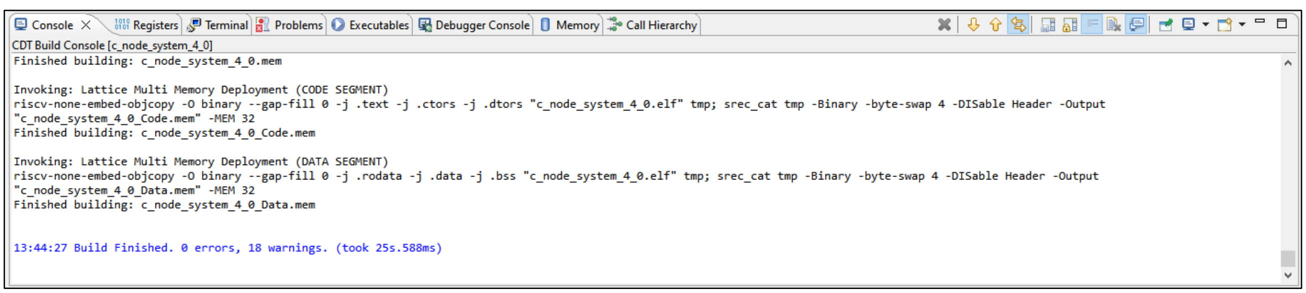


Figure C.33. Completing Process

12. To locate the binary file and .mem file to below path:
\NodeSystem\node_system_4_0\c_node_system_4_0\Debug.

C.4. Generating the Bit File in the Main System

C.4.1. Primary Main System

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

1. Open the Propel builder 2024.1 tool.
2. Click on the open design symbol and go to the below path:
Main_System\Primary_MainSystem\soc_main_system_4_0\soc_main_system_4_0.

3. If you 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.
4. Select the **soc_main_system_4_0.sbx** file and the design opens.

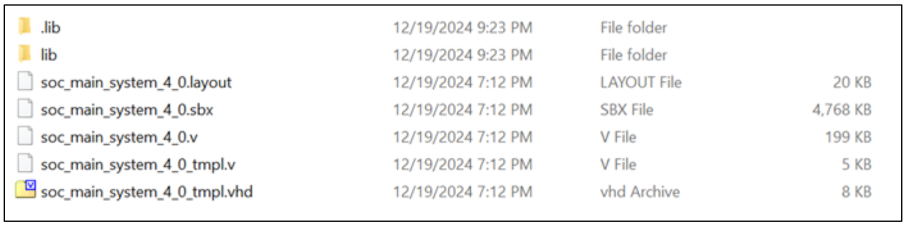


Figure C.34. soc_main_system.sbx

5. Double-click on the **system0_inst**. A pop-up window appears as shown in Figure C.35.

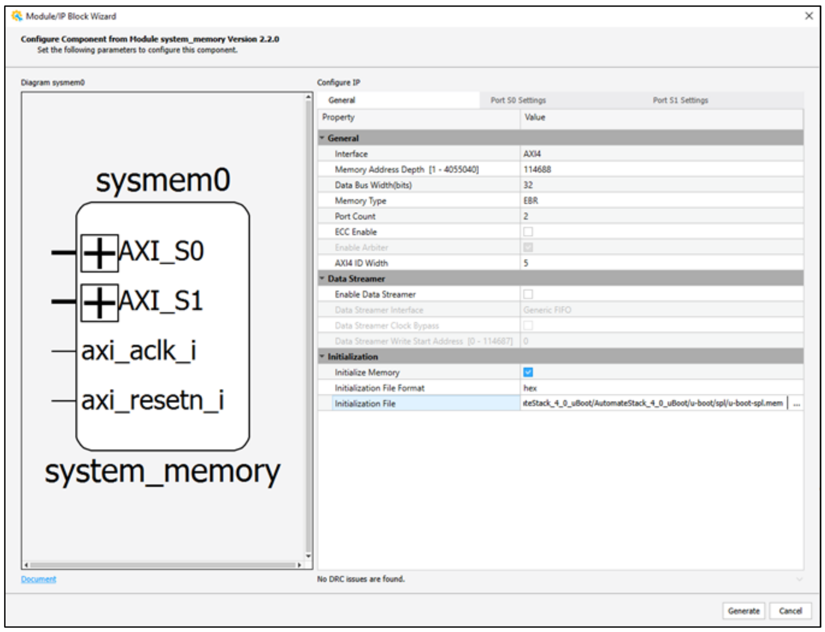


Figure C.35. System Initialization File

6. Initialize the data memory with the generated *u-boot-spl.mem* file in the **\AutomateStack_4_0_uBoot\u-boot\spl** folder of the AutomateStack_4_0_uBoot.
7. Click **Generate** and **Validate**.

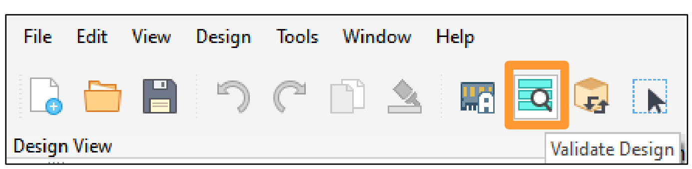


Figure C.36. Validate Button

8. Click **Generate SGE**.

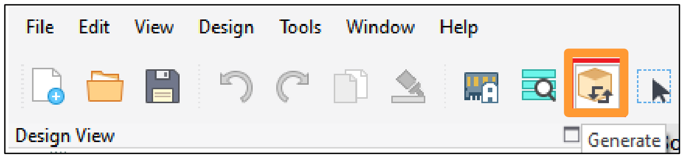


Figure C.37. Generate SGE Button

9. Open the Radiant tool in the Propel Builder interface.

Figure C.38. Radiant Tool Button

Note: No need to change the below settings just ensure that these settings are enabled.

a. Open the generated Radiant Project in the Radiant Tool:
\Main_System\Primary_MainSystem\soc_main_system_4_0

b. Select the **soc_main_system_4_0.rdf** file and the project opens.

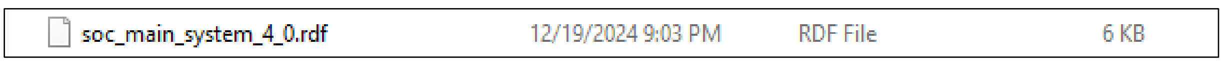


Figure C.39. soc_main_sysyem.rdf File

c. Double-click LAV-AT-E70ES1-3LFG1156.

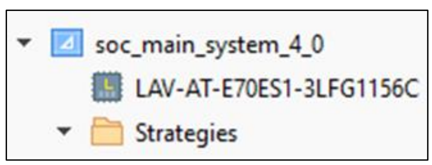


Figure C.40. LAV-AT-E70ES1-3LFG1156C

d. Select Family: **LAV-AT**

e. Select Device: **LAV-AT-E70ES1**

f. Select Operating Condition: **Commercial**

g. Select Package: **LFG1156**

h. Performance Grade: **9_High-Performance_1.0V**

i. Part Number: **LAV-AT-E70ES1-3LFG1156C**

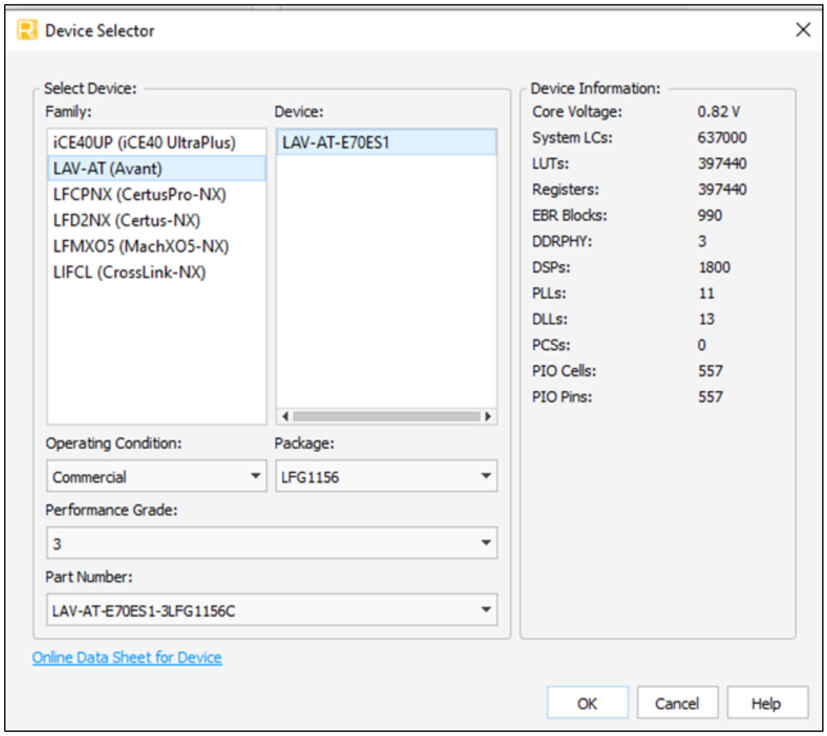


Figure C.41. Lattice Radiant Device Selector for Main System

j. Set Frequency parameter to **200 MHz**.

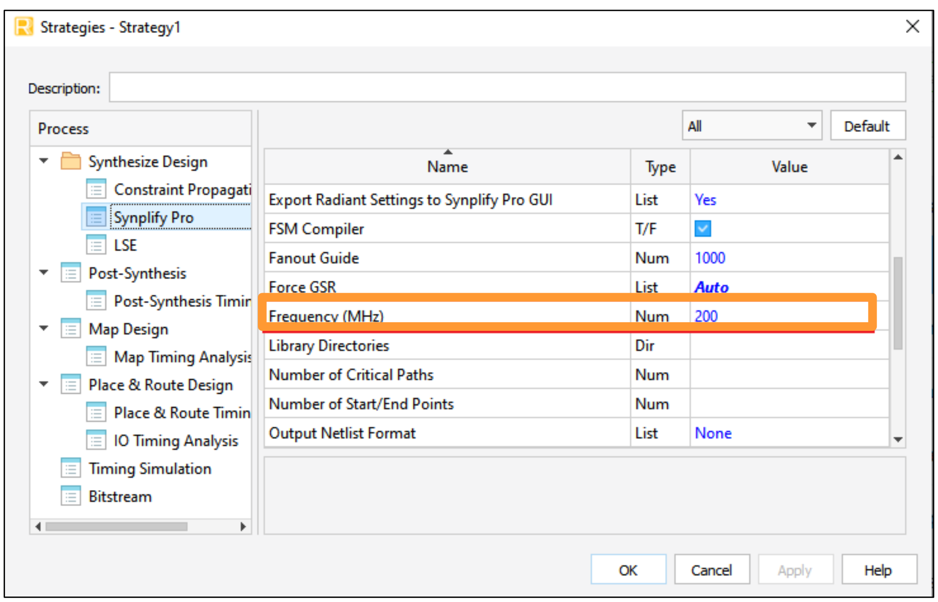


Figure C.42. Strategy for Build Generation for Main System

k. Go to the Strategy and select the **Map Design**.

l. Go to **Map Timing Analysis** and select the highlighted part as shown in Figure C.43.

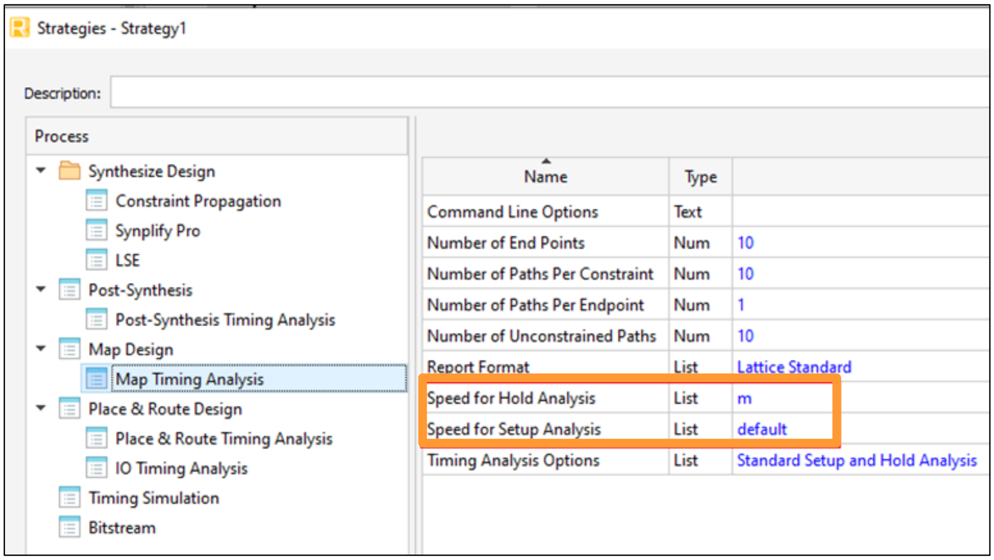


Figure C.43. MAP Analysis Setting for Main System Bit File Generation

m. Go to **Place & Route Design** and select the settings as shown in Figure C.44.

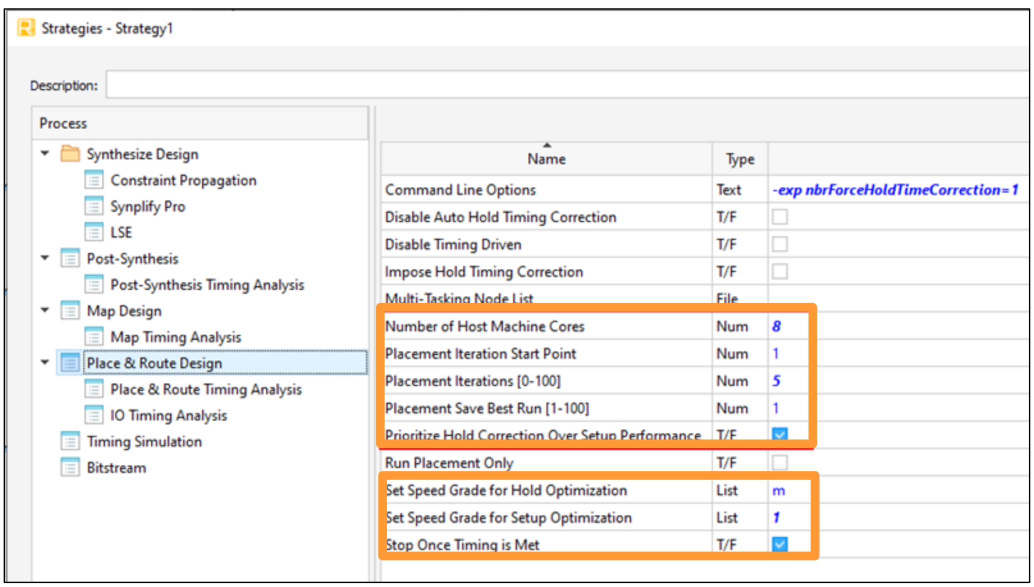


Figure C.44. PAR Setting for Main System Bit File Generation

n. Go to **Place and Route Timing Analysis** and select the settings shown in Figure C.45.

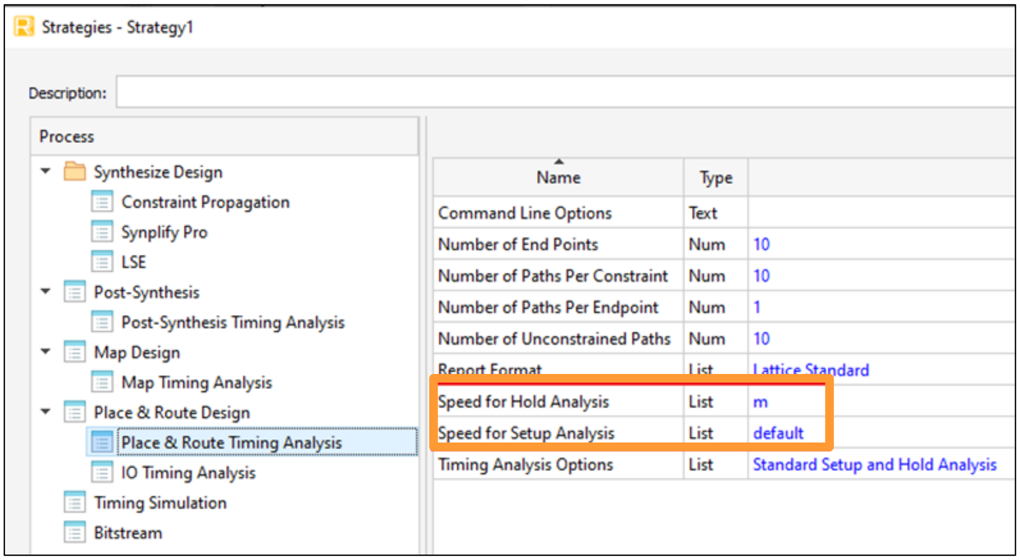


Figure C.45. PAR Timing Analysis Setting for Main System Bit File Generation

10. Go to Bitstream and select the **IP Evaluation** if you want to generate the non-licensed bit file. If you want to generate licensed bit file, uncheck the IP Evaluation box.

    Note: You need to request for license file from official website of Lattice Semiconductor.

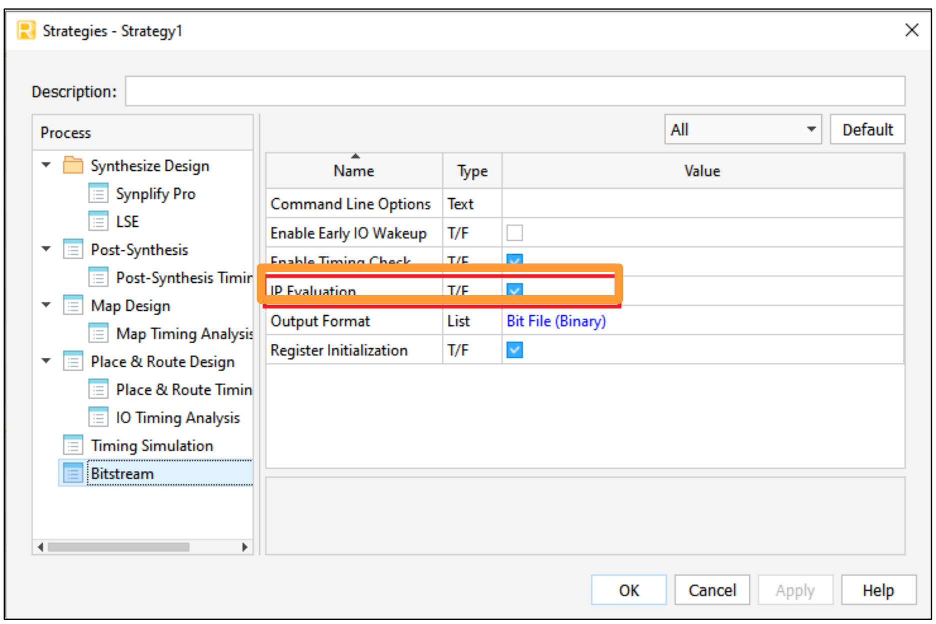


Figure C.46. IP Evaluation

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

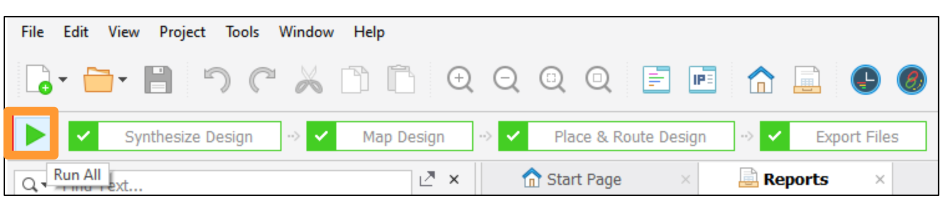


Figure C.47. Run All Button

12. To locate the bit stream file follow the below path: **\Main_System\Primary_MainSystem\soc_main_system_4_0\impl_1**.

soc_main_system_4_0_impl_1.bgn	12/19/2024 8:22 PM	BGN File	11 KB
soc_main_system_4_0_impl_1.bit	12/19/2024 8:22 PM	BIT File	12,928 KB

Figure C.48. Bitstream File

C.4.2 Golden Main System

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

1. Open the Propel builder 2024.1 tool.
2. Click on the open design symbol and go to the below path: **\Main_System\Golden_MainSystem\soc_main_system_3_1\soc_main_system_4_0**.
3. If you do not have the Propel patch, open directly from where project is located. Make sure that there no space in the folder name.
4. Select the **soc_main_system_4_0.sbx** file to open the design.

.lib	12/19/2024 9:23 PM	File folder	
lib	12/19/2024 9:23 PM	File folder	
soc_main_system_4_0.layout	12/19/2024 7:12 PM	LAYOUT File	20 KB
soc_main_system_4_0.sbx	12/19/2024 7:12 PM	SBX File	4,768 KB
soc_main_system_4_0.v	12/19/2024 7:12 PM	V File	199 KB
soc_main_system_4_0_tmpl.v	12/19/2024 7:12 PM	V File	5 KB
soc_main_system_4_0_tmpl.vhd	12/19/2024 7:12 PM	vhd Archive	8 KB

Figure C.49. soc_main_system.sbx

5. Double-click on the system0_inst. A pop-up window appears as below.

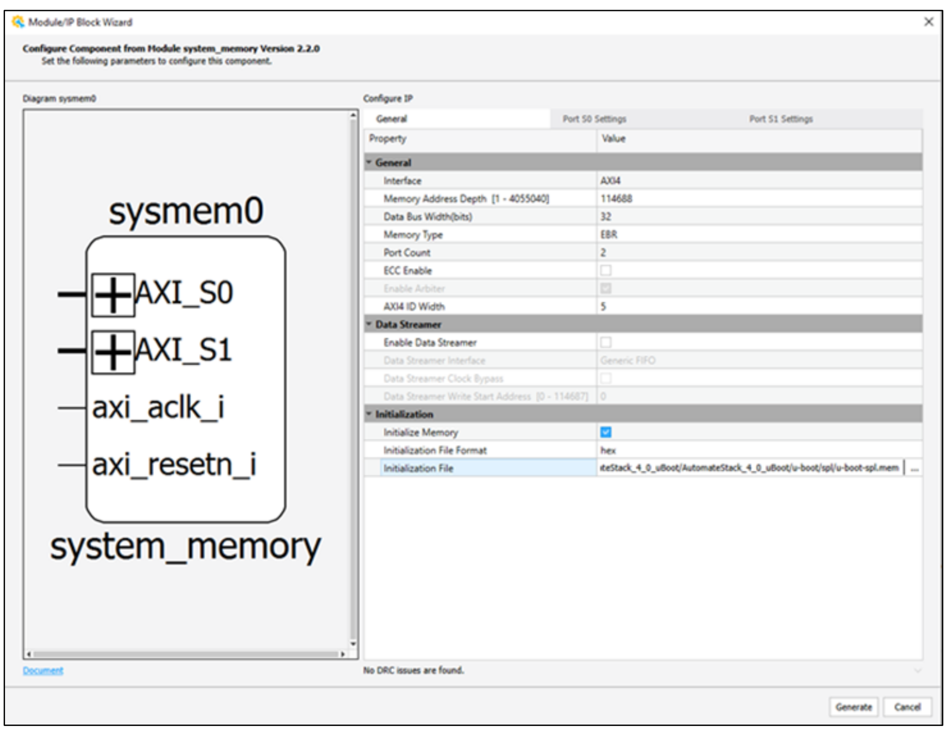


Figure C.50. System Initialization File

6. Initialize Data memory with generated *u-boot-spl.mem* file in the **AutomateStack_4_0_uBoot\u-boot\spl** folder of AutomateStack_4_0_uBoot.
7. Click **Generate** and **Validate**.

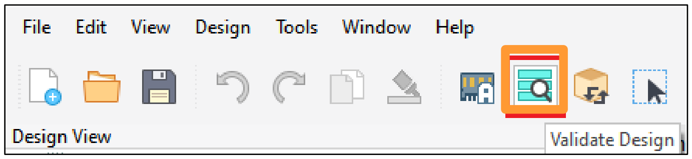


Figure C.51. Validate Button

8. Click the **Generate SGE** button.

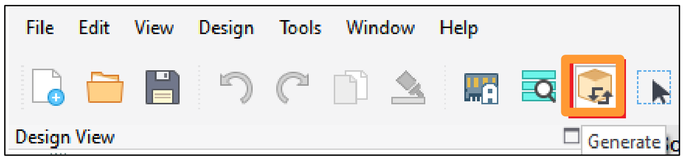


Figure C.52. Generate SGE Button

9. Open the Radiant tool in the Propel builder interface.

Figure C.53. Radiant Tool Button

Note: No need to change the below settings; just ensure that these settings are enabled.

a. Open the generated Radiant Project in the Radiant Tool: **Main_System\Golden_MainSystem\soc_main_system_4_0**.
b. Select the **soc_main_system_4_0.rdf** file and the project opens.

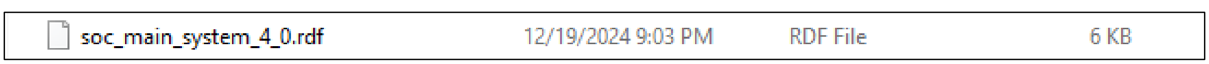


Figure C.54. soc_main_sysyem.rdf file

c. Double-click LAV-AT-E70ES1-3LFG1156.

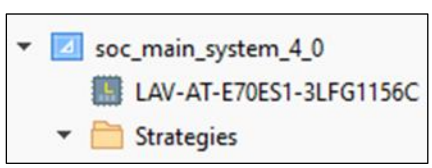


Figure C.55. LAV-AT-E70ES1-3LFG1156C

d. Select Family: **LAV-AT**
e. Select Device: **LAV-AT-E70ES1**
f. Select Operating Condition: **Commercial**
g. Select Package: **LFG1156**

h. Performance Grade: **9_High-Performance_1.0V**

i. Part Number: **LAV-AT-E70ES1-3LFG1156C**

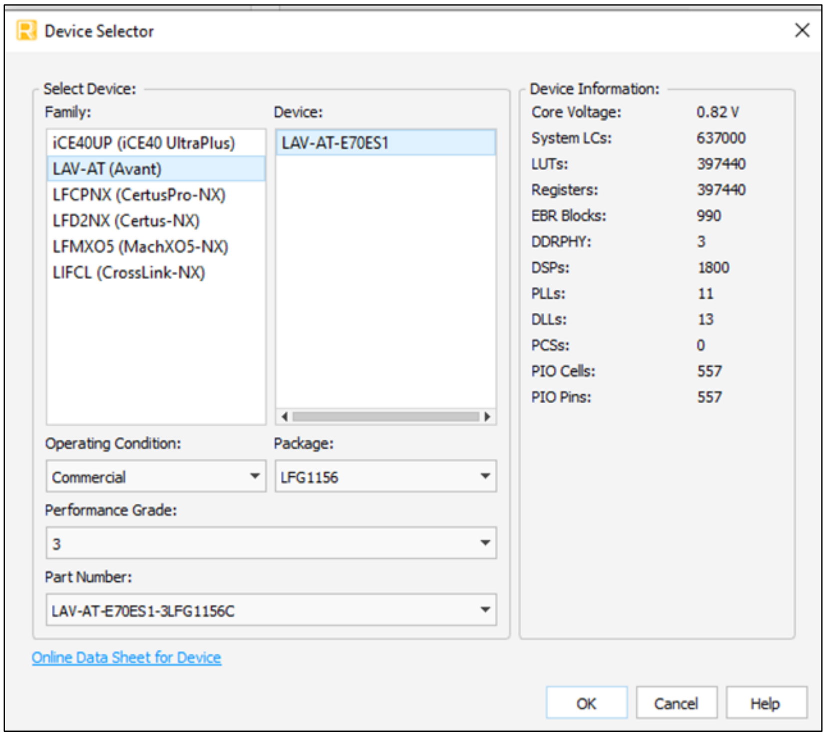


Figure C.56. Lattice Radiant Device Selector for Main System

j. Set Frequency parameter to **200 MHz**.

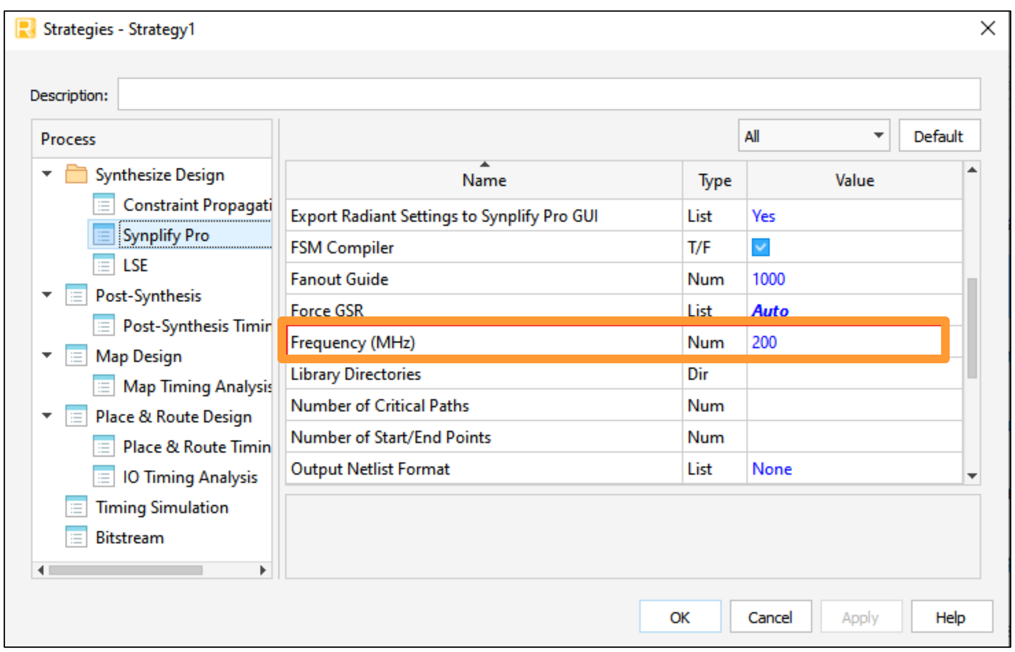


Figure C.57. Strategy for Build Generation for Main System

k. Go to the Strategy and select the Map Design and select the **Map Timing Analysis**. Apply the settings shown in Figure C.58.

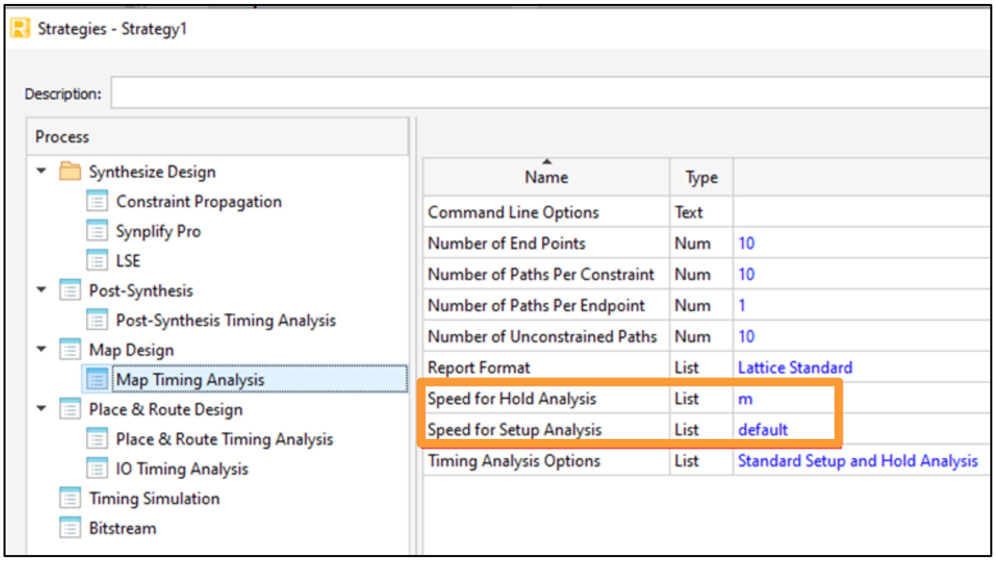


Figure C.58. MAP Analysis Setting for Main System Bit File Generation

l. Go to **Place & Route Design** and select the settings shown in Figure C.59.

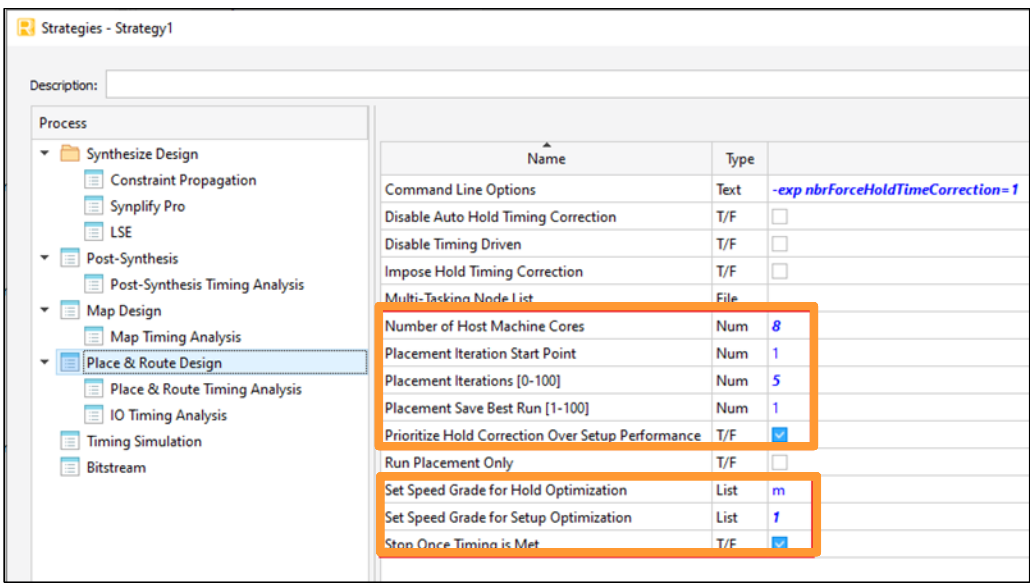


Figure C.59. PAR Setting for Main System Bit File Generation

m. Go to **Place and Route Timing Analysis** and select the settings shown in Figure C.60.

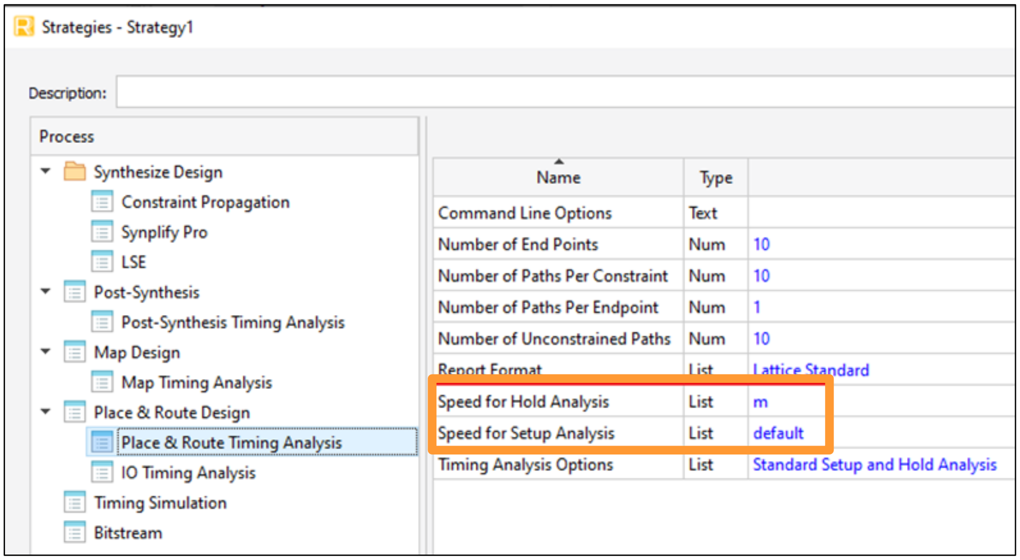


Figure C.60. PAR Timing Analysis Setting for Main System Bit File Generation

10. Go to Bitstream and select the **IP Evaluation** if you want to generate the non-licensed bit file. If you want to generate licensed bit file, uncheck the IP Evaluation box.

    Note: You need to request for license file from official website of Lattice Semiconductor.

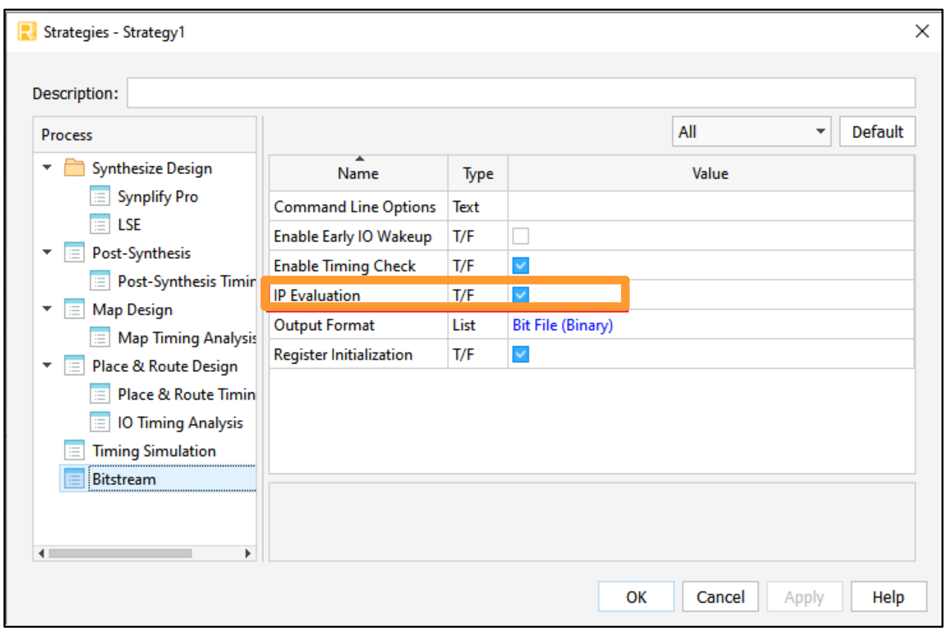


Figure C.61. IP Evaluation

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

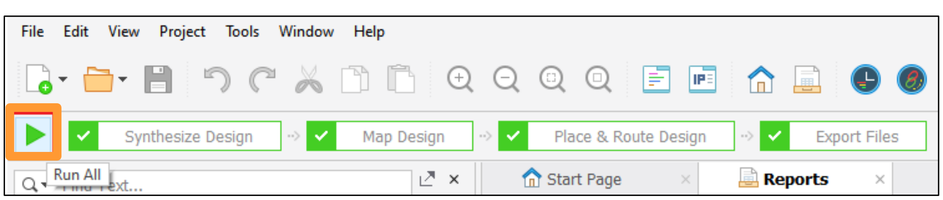


Figure C.62. Run All Button

12. To locate the bit stream file, follow the below path:
\Main_System\Golden_MainSystem\soc_main_system_4_0\impl_1.

soc_main_system_4_0_impl_1.bgn	12/19/2024 6:32 PM	BGN File	11 KB
soc_main_system_4_0_impl_1.bit	12/19/2024 6:32 PM	BIT File	12,928 KB

Figure C.63. Bitstream File

C.4.3. Node System

1. Open the Propel builder 2024.1 tool.
2. Click on the open design symbol and go to the below path:
NodeSystem\node_system_4_0\soc_node\soc_node.

Name	Date modified	Type	Size
.lib	14-12-2023 10:06 PM	File folder	
application	14-12-2023 05:00 PM	File folder	
lib	15-12-2023 10:51 AM	File folder	
soc_node.layout	15-12-2023 10:51 AM	LAYOUT File	9 KB
soc_node.sbx	15-12-2023 10:51 AM	SBX File	2,076 KB

Figure C.64. soc_node.sbx

3. Double-click on the system0_inst. A pop-up window appears on the screen as shown below.

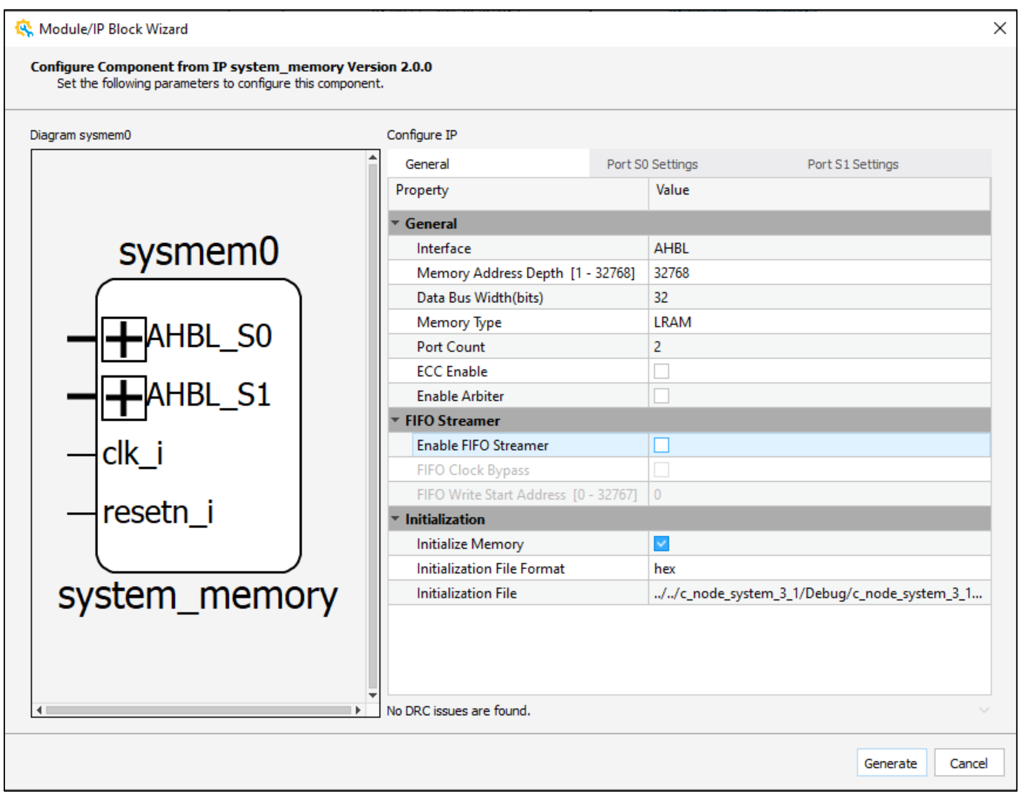


Figure C.65. System0 Initialization

4. Initialize the data memory with the generated *c_node_system_4_0_Data.mem* file in debug folder of C project.

5. Click **Validate**.

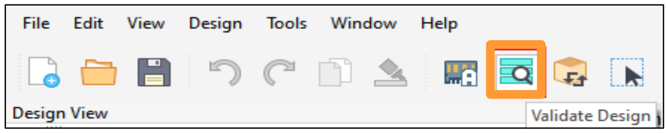


Figure C.66. Validate Button

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

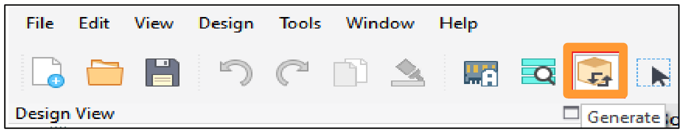


Figure C.67. Generate SGE Button

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

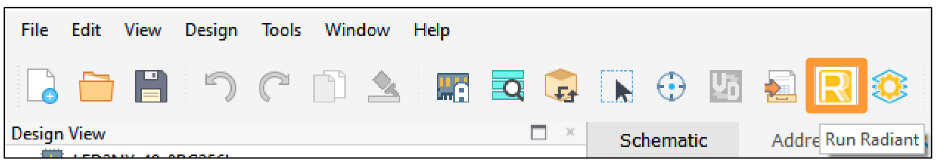


Figure C.68. Radiant Tool Button

Note: No need to change the below settings; just ensure that these settings are enabled.

a. Open the generated Radiant Project in the Radiant Tool: **NodeSystem\node_system_4_0\soc_node.**

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

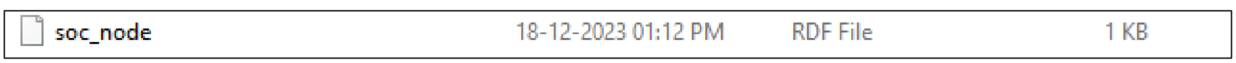


Figure C.69. soc_node.rdf file

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

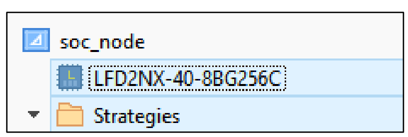


Figure C.70. LFD2NX-40-8BG256C

d. Select Family: **LFD2NX**

e. Select Device: **LFD2NX-40**

f. Select Operating Condition: **Commercial**

g. Select Package: **CABGA256**

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

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

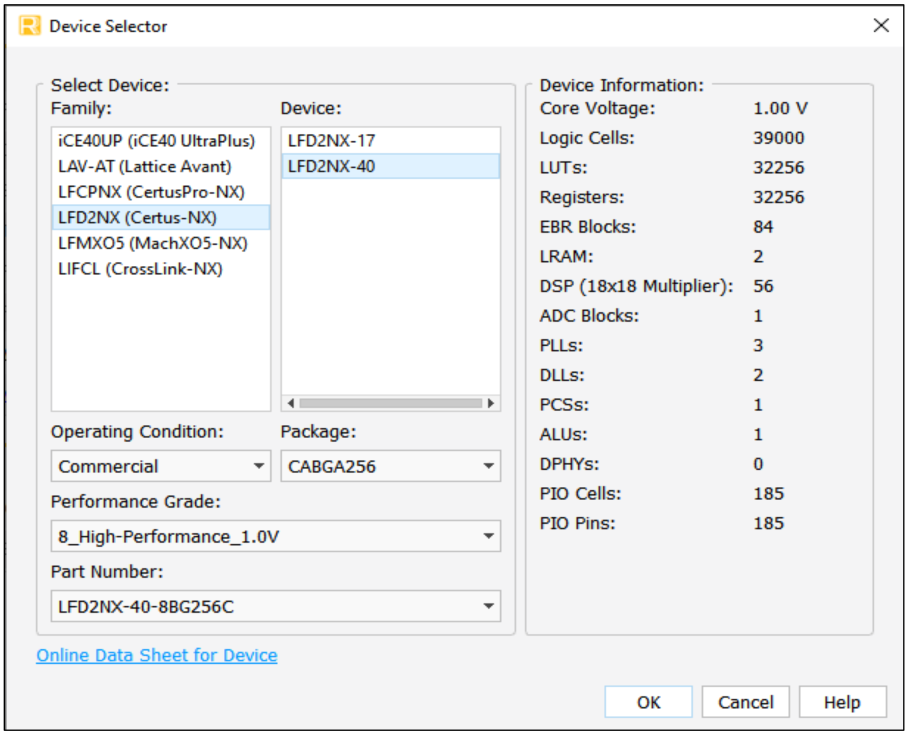


Figure C.71. Lattice Radiant Device Selector for Node System

j. Set Frequency parameter to **150 MHz**.

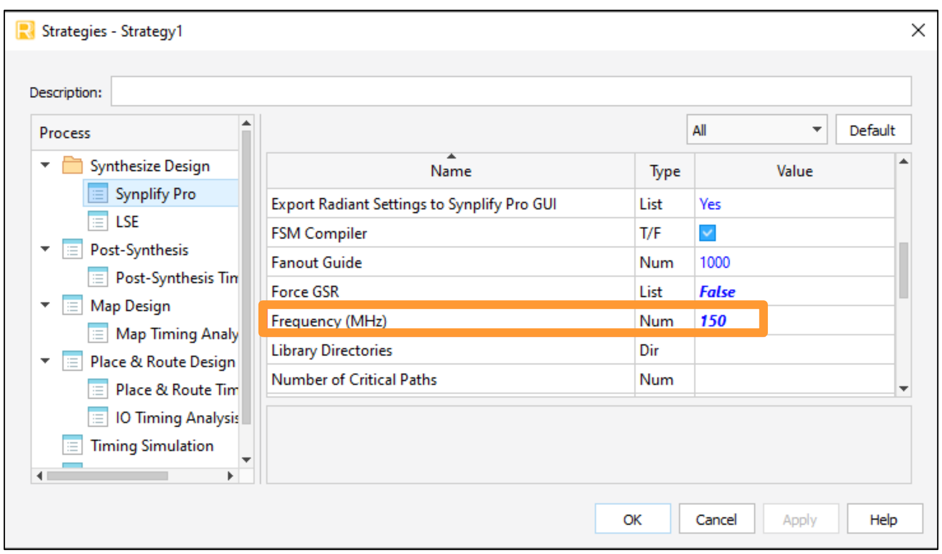


Figure C.72. Strategy for Build Generation for Node System

k. Go to the Strategy and click the **Map Design**. Select the **Map Timing Analysis** and apply the settings shown in Figure C.73.

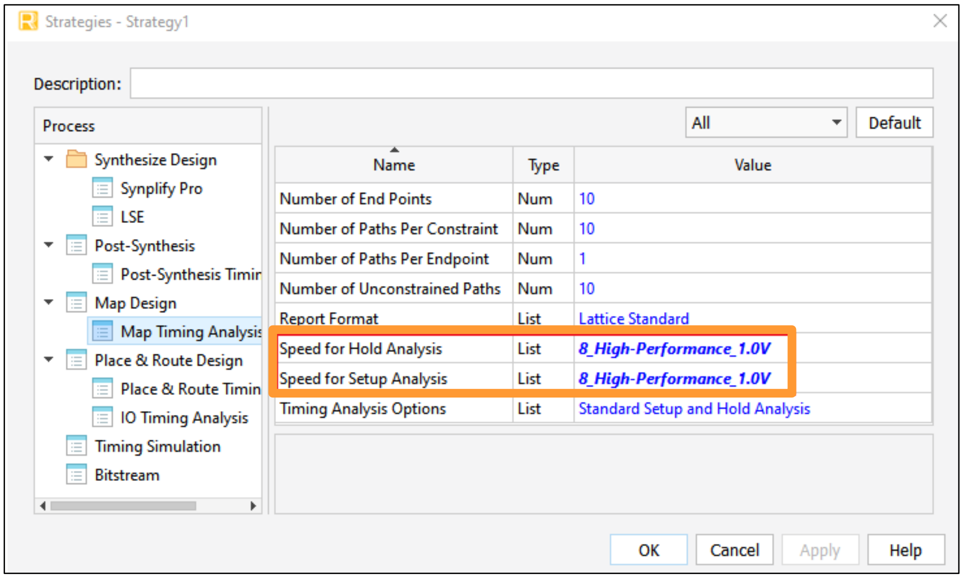


Figure C.73. MAP Analysis Setting for Node System Bit File Generation

l. Go to **Place & Route Design** and select the settings shown below.

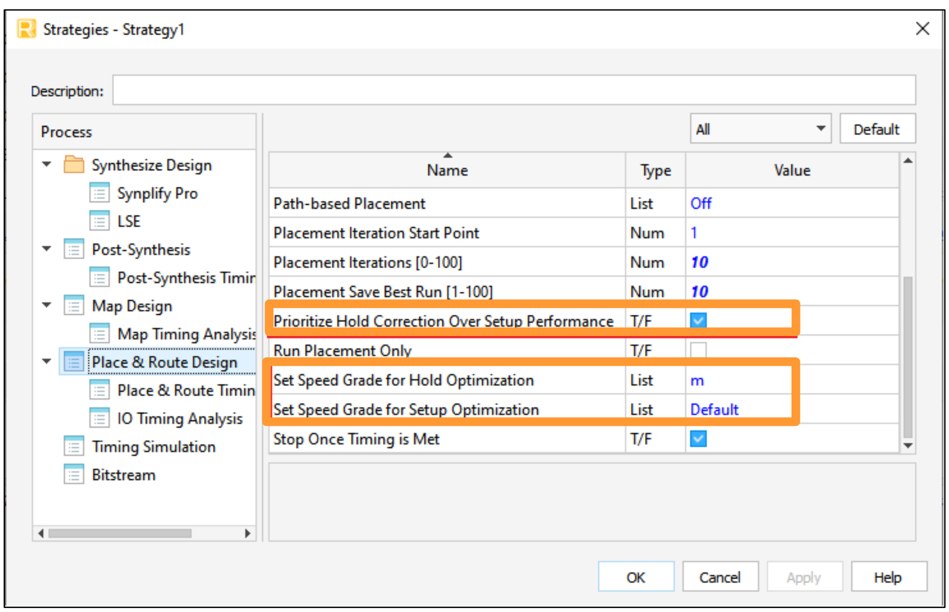


Figure C.74. PAR setting for Node System Bit File Generation

m. Go to **Place and Route Timing Analysis** and select the settings shown in Figure C.75.

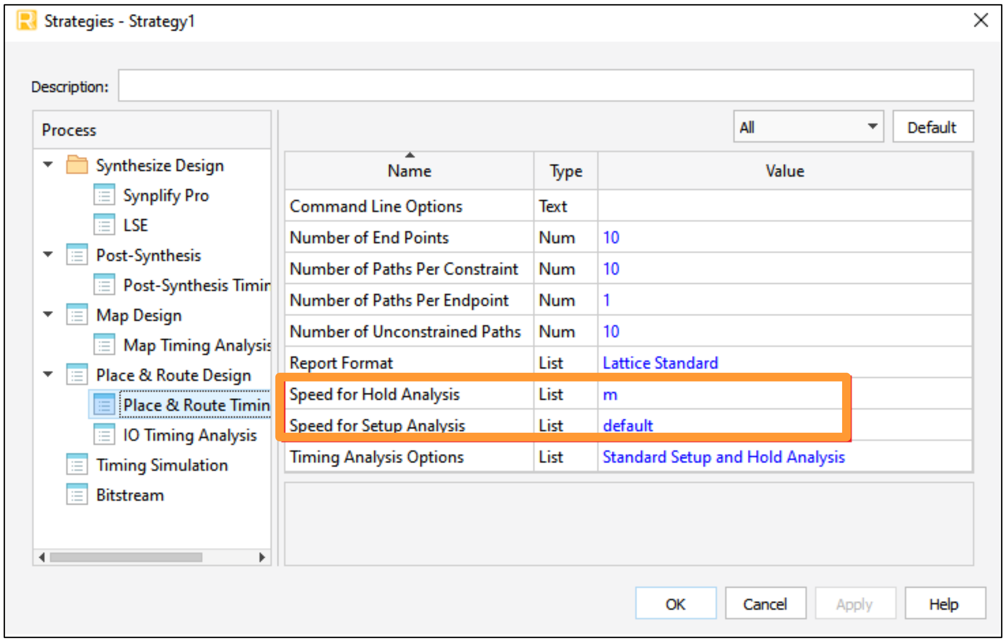


Figure C.75. PAR Timing Analysis Setting for Node System Bit File Generation

8. Go to Bitstream and select the IP Evaluation if you want to generate non-licensed bit file. If you want to generate the licensed bit file, uncheck the IP Evaluation box.

   Note: You need to request for license file from official website of Lattice Semiconductor.

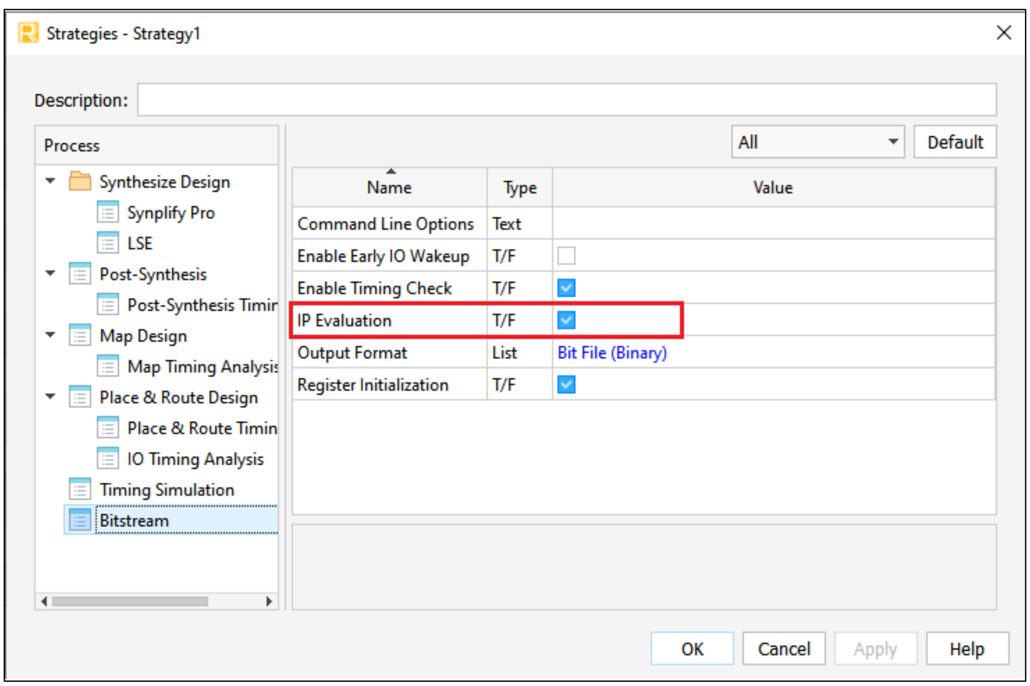


Figure C.76. IP Evaluation

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

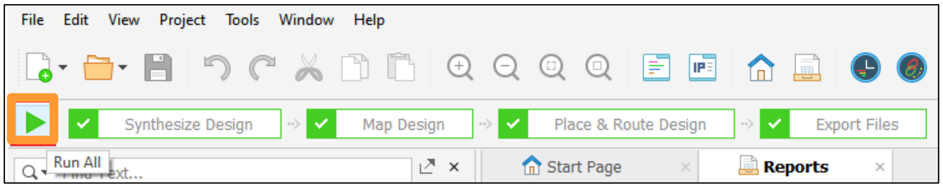
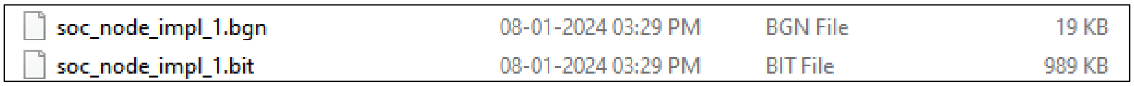


Figure C.77. Run All Button

10. To locate the bit stream file, follow the below path: **\NodeSystem\node_system_4_0\soc_node\impl_1**.



soc_node_impl_1.bgn	08-01-2024 03:29 PM	BGN File	19 KB
soc_node_impl_1.bit	08-01-2024 03:29 PM	BIT File	989 KB

Figure C.78. Bitstream File

Appendix E. Creating the MCS File

The following steps provide the procedure 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, Alternate pattern 1.

To create the MCS file, perform the following:

1. Open the Lattice Radiant Programmer > Tools > Deployment Tool.

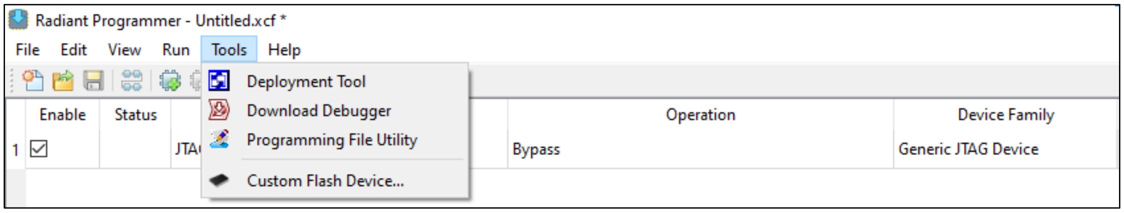


Figure E.1. Deployment Tool

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

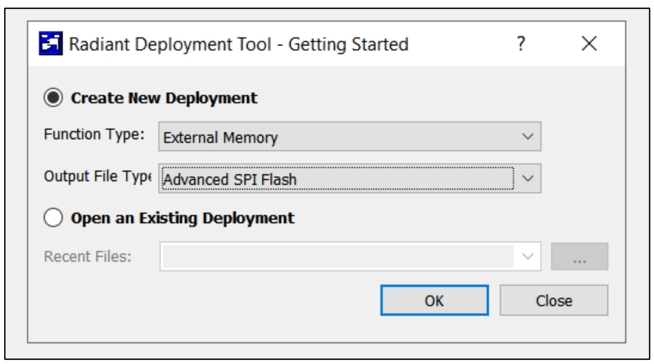


Figure E.2. Creating New Deployment for Multi-Boot

4. For **Step 1 of 4: Select input files** window, apply the settings below.
   a. 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.
   b. Select **Next**.

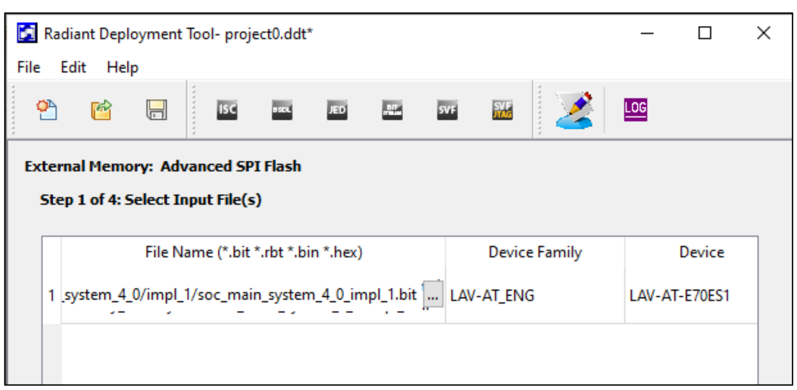


Figure E.3. Select Input File Window

5. For **Step 2 of 4: Advanced SPI Flash Options** window, apply the settings below.
    a. Go to the **Multiple Boot** tab.
    b. Select the Multi-Boot option.
    c. Click on the Golden pattern browse button to select the Primary pattern bitstream.
    d. The starting address of the Golden pattern is automatically assigned. You can change it by clicking on the drop-down menu.
    e. In the number of Alternate patterns field, select the number of patterns to include through the drop-down menu.
    f. 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.
    g. 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.
    h. Select **Next**.

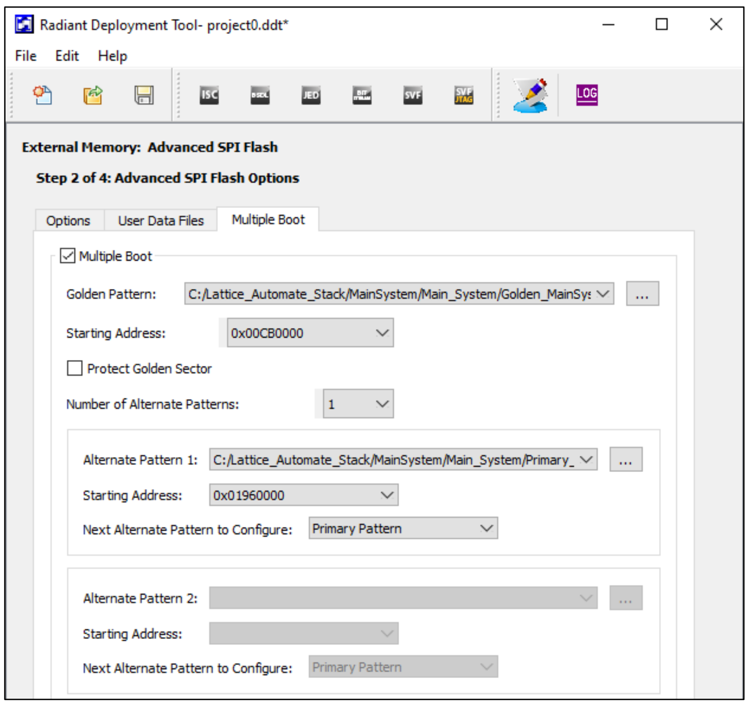


Figure E.4. Advanced SPI Flash Options - Multi-Boot Tab Window

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

6. For **Step 3 of 4: Select output file window**, apply the settings below.
    a. Specify the name of the output PROM hex file in the output file 1 field.
    b. Select **Next**.

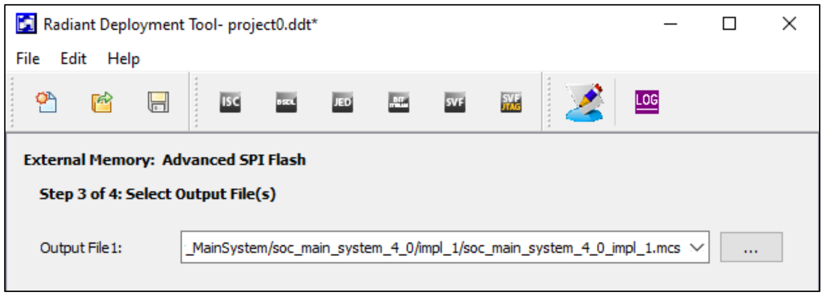


Figure E.5. Select Output File Window

7. For **Step 4 of 4: Generate Deployment** window, apply the settings below.
   a. Review the summary information.
   b. If everything is correct, click **Generate**. The generate deployment pane indicates the PROM file is successfully generated.
   c. Save the deployment setting by selecting **File > Save**.
   d. To exit, go to **File > Exit**.

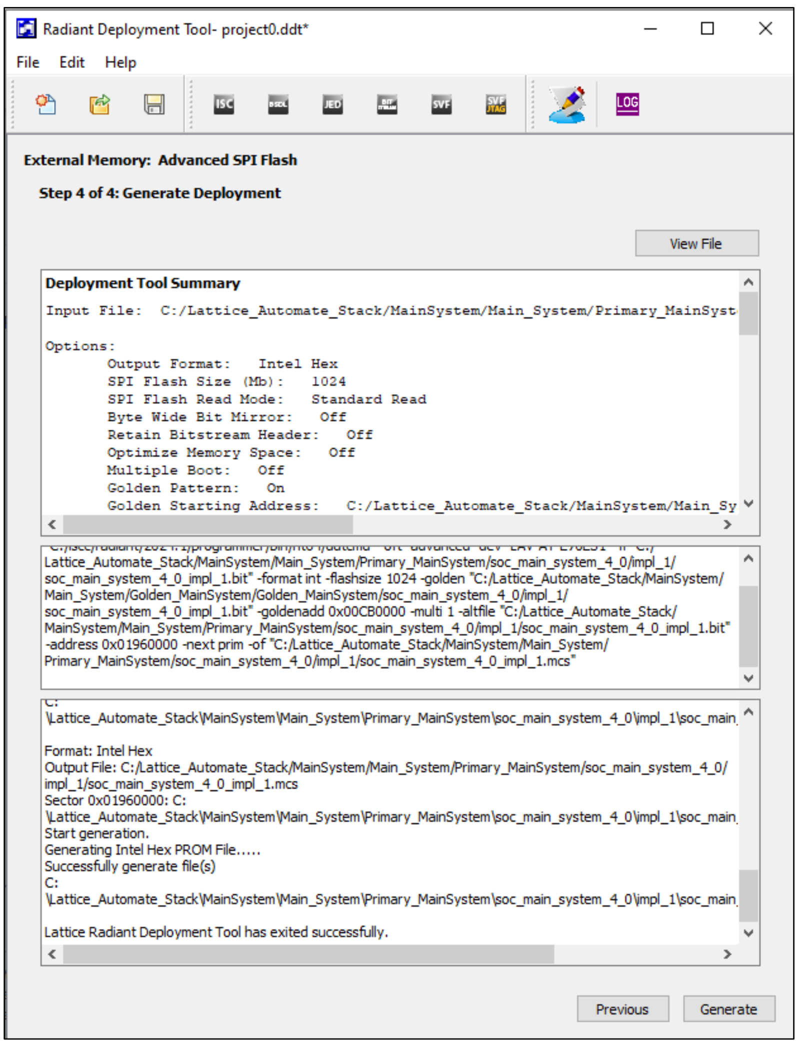


Figure E.6. Generate Deployment window

8. Once configured, you can program the .mcs file in the external flash using the Radiant Programmer.

References

- Lattice Automate

Other references:

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

Technical Support Assistance

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

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

Revision History

Revision 1.0, February 2025

Section	Change Summary
All	Initial preliminary release.

LATTICE
SEMICONDUCTOR®

www.latticesemi.com