Automate Stack 3.0

Reference Design

FPGA-RD-02267-1.0

March 2023

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 associated risk the responsibility entirely of the Buyer. Buyer shall not rely on any data and performance specifications or parameters provided herein. 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. No Lattice products should be used in conjunction with mission- or safety-critical or any other application in which the failure of Lattice's product could create a situation where personal injury, death, severe property or environmental damage may occur. 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.

Contents

Figures

Tables

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
AI	Artificial intelligence
AXI4	Advanced eXtensible Interface Architecture 4
BLDC	Brushless DC
CCU	CNN Co-processor Unit
CNN	Convolutional Neural Network
DDR	Double Data Rate
DL	Deterministic Latency
DMA	Direct Memory Access
Etherconnect	Light weight packet-based protocol similar to ethernet
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GPIO	General-Purpose Input/Output
GUI	Graphical User Interface
I2C	Inter-Integrated Circuit
IP address	Internet Protocol address
IPV4	Internet Protocol Version 4
ISR	Interrupt Service Routine
MAC	Media Access Control Address
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
MS	Main System
NS	Node System
OPC	Open Platform Communications
OPCUA	Open Platform Communications Unified Architecture
OS	Operating System
PCIe	Peripheral Component Interconnect express
PDA	Percent Defective Allowable
PDM	Predictive Maintenance
PMOD	Peripheral Module Interface
QSPI	Quad Serial Peripheral Interface
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RPM	Revolutions Per Minute
SFP	Small Form Factor Pluggable
SGMII	Serial Gigabit Media Independent Interface
SOC	System on Chip
SPI	Serial Peripheral Interface
UA	Unified Architecture
UART	Universal Asynchronous Receiver-Transmitter
UADP	Unified Access Data Plane
UDP	User Datagram Protocol

1. Introduction

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

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

Lattice Automate Stack 1.1 supports two chains of nodes that can be connected to 1 main system board. All the nodes are synchronized physically. Main system supports dynamic pulse-based system synchronization scheme, in which it checks nodes disconnection during runtime and compensates clock ppm to calculate synchronization delay. It supports OPC UA server/client-based user interface which is running on host PC and client is running on Raspberry Pi board.

Lattice Automate Stack 2.0 supports all features of Lattice Automate Stack 1.1. It supports MQTT broker/client-based host application, Python Interface as host control, and also supports PCIe® interface as host for high-speed applications. In the node side, it has motor IP for motor-based features and has standard SPI Manager and I2C Manager interfaces to connect various peripherals (sensors) into system.

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

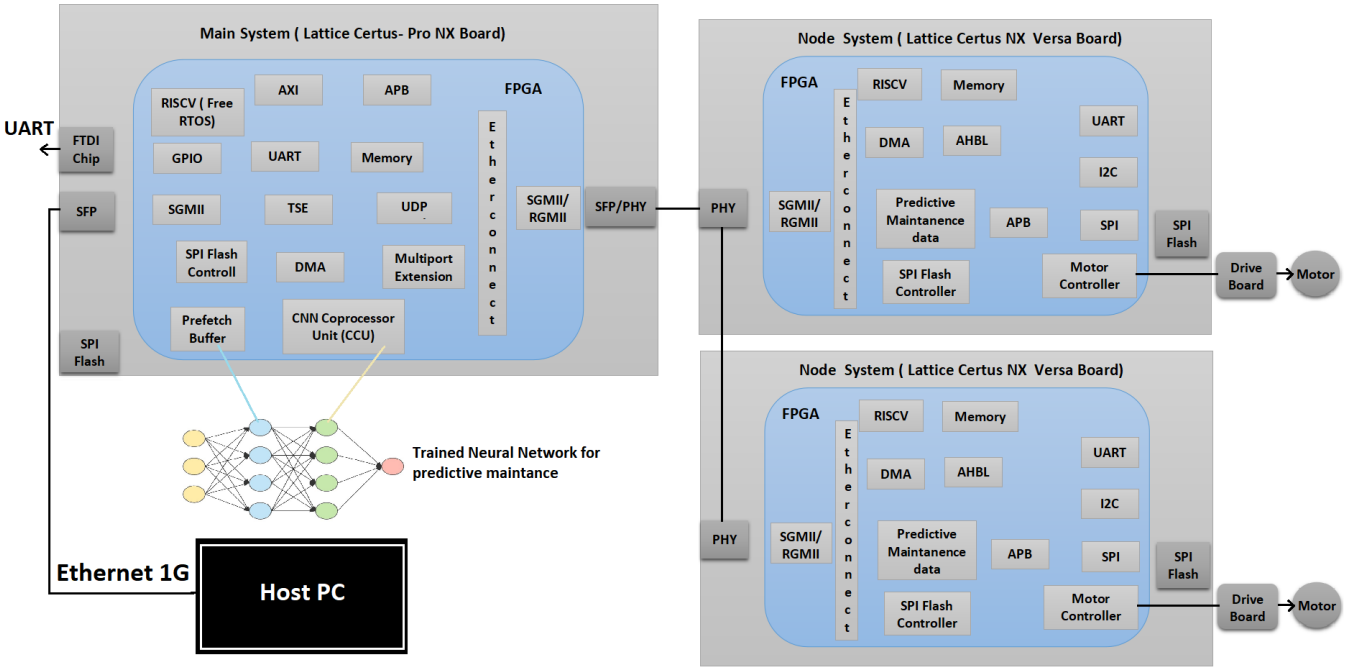

Figure 1.1. Top Level Block Diagram of Automate Stack 3.0

1.1. Components

The Automate Stack 3.0 release includes the following components:

- System on Chip (SOC)
  - Main System IPs
    - EtherControl IP (With SGMII/RGMII (phy or sfp)), FIFO DMA, CNN Co-Processor Unit (CCU), SPI Flash Controller, Multiport extension, UDP Stack, SGMII TSE MAC, and Reset Synchronizer.
  - Node System IPs
    - EtherControl IP (With SGMII/RGMII (phy or sfp)), FIFO DMA, BLDC motor control IP, and 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).
  - User Interface
    - Controls motor, collects status and data for predictive maintenance, and displays a warning when maintenance is 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 3.0 consists of one Main System (MS) with ether connect main and multiple Node Systems (NS). The host will be connected to the MS through Ethernet cable. Application software with GUI running on host can send commands to the MS and receive motor maintenance data from the system for AI training. The MS can propagate the commands to NS for motor control and gather maintenance data from NS.

The Certus™-NX versa board and CertusPro™-NX versa board are used for basic demo of complete system.

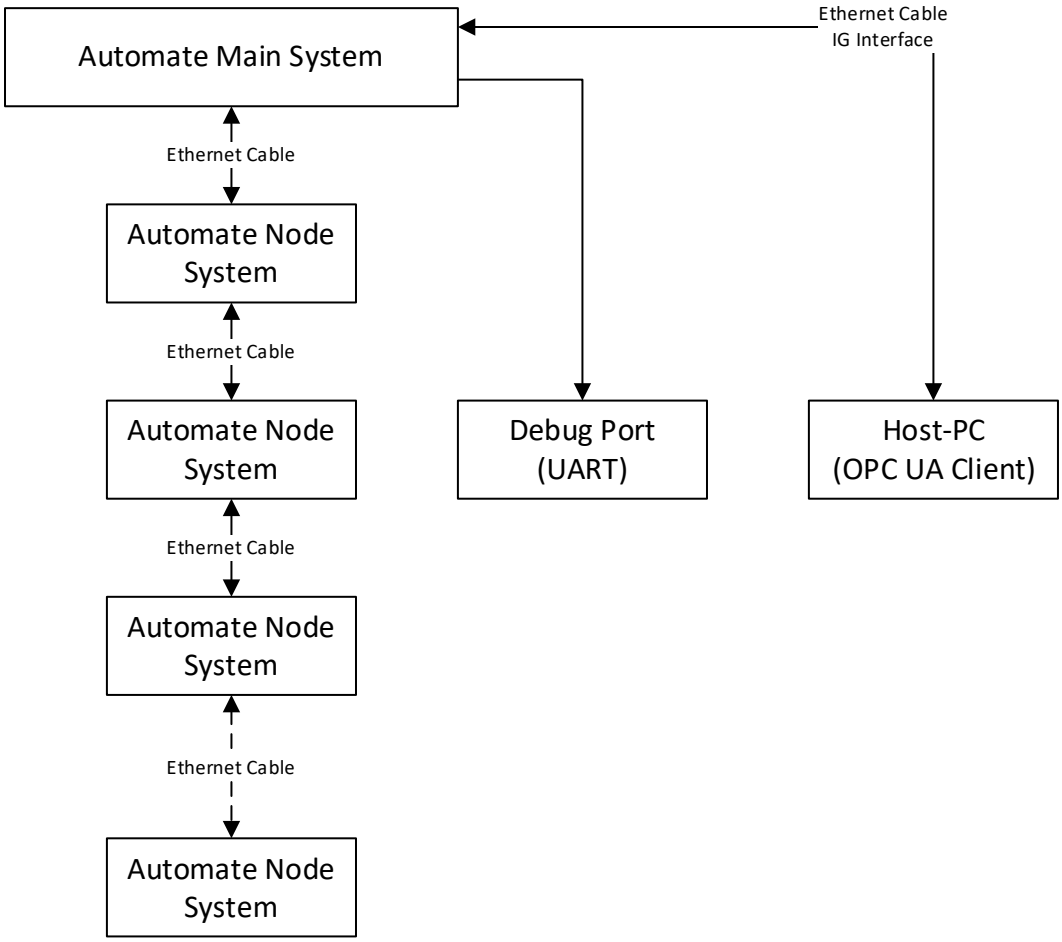

Figure 2.1. Automate Stack 3.0 Architecture

2.2. FPGA Design

2.2.1. Main System

The Main System architecture is shown in Figure 2.2.The AXI4 Interconnect has four managers and nine subordinates:

The following are the AXI4 Interconnect managers:

- RISC-V CPU Instruction Cache
- RISC-V CPU Data Cache
- CNN Co-Processor Unit
- FIFO DMA

The following are the AXI4 Interconnect subordinates:

- ISR RAM
- Data RAM (port S0 and S1)
- CNN Co-Processor Unit
- FIFO DMA
- Ether Control
- AXI2APB Bridge
- Multiport Extension
- QSPI Memory Controller with pre-fetch buffer (SPI Flash Controller)

The free RTOS(RISC-V) CPU, CNN Co-processor Unit (CCU), and FIFO DMA can access data to the shared memory Data Ram, Ether Control, SPI Flash Controller, FIFO DMA, CNN Co-processor Unit (CCU), Multiport Port Extension and AXI2APB bridge directly and UDP IP, SGMII TSE MAC Wrapper, Multiport Extension & GPIO through AXI2APB bridge. Multiport Extension, GPIO, Ether Control can generate interrupts to RISC-V CPU.

For performance and nearly deterministic latency (DL), it uses port S0 of the Data RAM exclusively for RISC-V CPU access. The other two managers, CNN Co-Processor Unit and FIFO DMA, access port S1 of the Data RAM. This way, the contention is avoided.

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

For better performance and nearly deterministic latency, Ether Control port supports one physical interface and it allows system to maintain a chain of nodes supporting up to 8 nodes.

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

- During boot, the boot loader copies the ISR code from the external flash to internal ISR RAM. It then sets up the ISR function pointer to this internal memory address.
- The ISR code is integrated in the bitstream and firms the ISR code in the ISR RAM as ROM code.

The first option can be used during initial development for debugging. The second option can be used in the final production release since it does not increase any system boot time.

The system is working at CPU frequency of 80 (System may be tuned to 100 MHz as well in upcoming release) MHz, the ethernet MAC protocol is working at 125 MHz and DDR Interface is working at 133 MHz

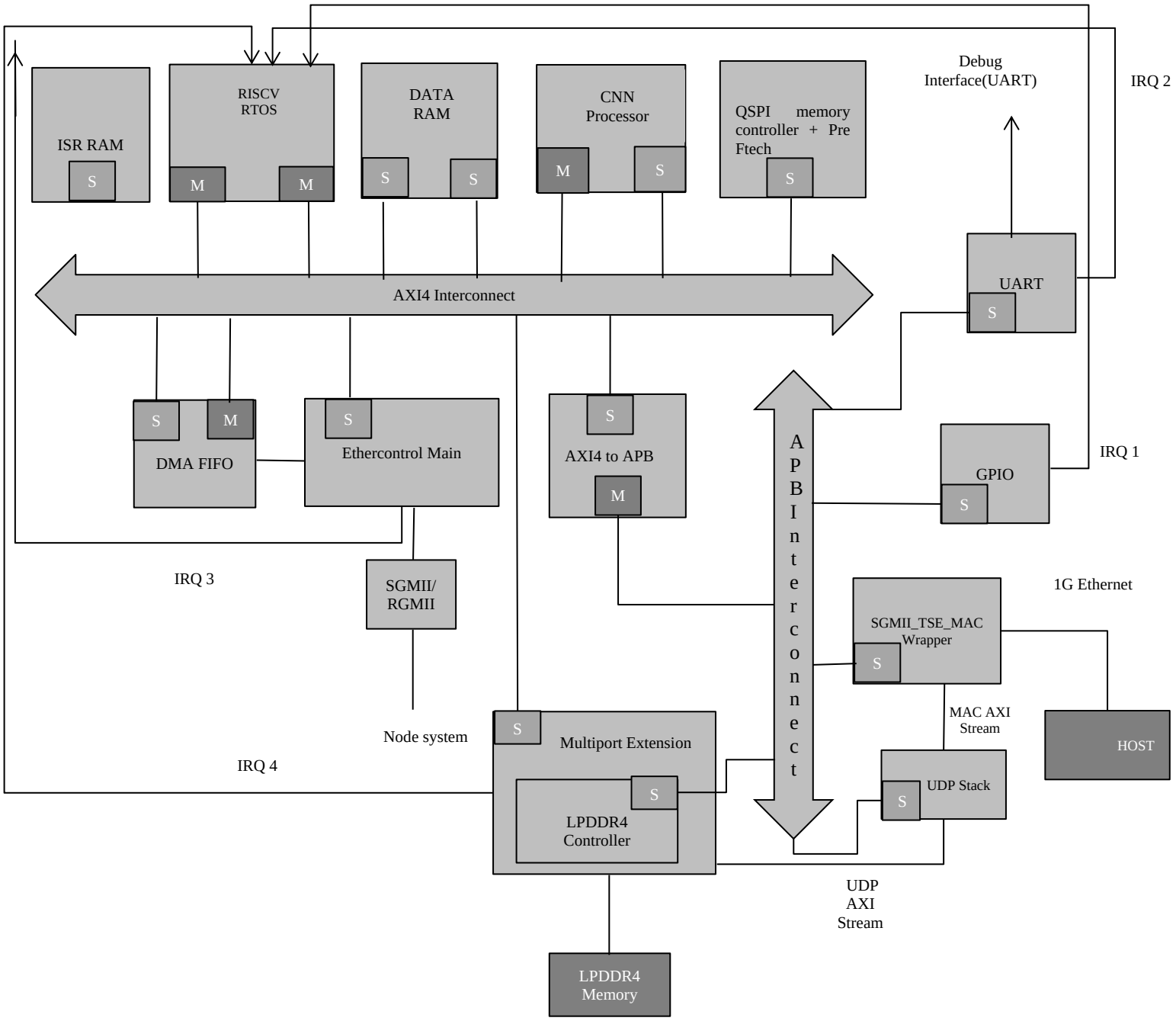

Figure 2.2. Main System Architecture

The firmware binary is stored in the external QSPI flash. When RISC-V at the Main System boots up, it sets the registers at QSPI Memory Controller (SPI Flash Controller). Then RISC_V jumps to the loaded firmware and executes the binary.

The RISC-V CPU fetches the commands and data from the host through DDR Interface(Channel one) by accessing AXI4 port of Multiport Extension IP through the AXI4 Interconnect.

RISC-V CPU sets the registers inside the CNN Co-Processor Unit and starts PDA operation. RISC-V CPU polls another register in the CNN Co-Processor Unit to check its operation status. RISC-V CPU requests for the new data for predictive maintenance from the subordinate PDM data collector by sending instructions through EtherControl IP. The data received from the

subordinate through Ethercontrol is transferred to data memory with DMA operation or sent to the host PC through Ethercontrol.

OPCUA UADP packet is created for the publisher and sent to the subscriber, it received the packet using ether control and its payload send to the UDP IP. UDP payload is further written to LPDDR4 and interrupt triggered to RISC-V, RISC-V sends read request to multiport extension IP. Top packets are stored in a FIFO. RISC-V fetches data from FIFO and passes to opcua software module. OPCUA software module decodes the UADP packet and extract RFL command. RFL packets are created and sent to the node system over ether control interface, which performs packetization and sends them to the downstream Node System. Node system takes the specific action according to the RFL command (RFL packet).

The information is written/read to/from peripherals connected to Nodes through SPI Manager/I2C Manager/ Modbus same as information is written (config)/read to (status)/from motor control IP.

RISC-V CPU gathers predictive maintenance data from downstream Node Systems through EtherControl and sends this data to the host PC from DDR Memory (Channel two) through AXI4 port. RISC-V CPU reads data from EtherControl through its AXI4 subordinate port, performs data processing, stores the data in the Data RAM, and then sends it to UART through APB. Alternatively, EtherControl can send downstream data to the FIFO DMA through its FIFO port, and FIFO DMA can write the data to the Data RAM directly.

At the end of every predictive maintenance cycle in software running on RISC-V, an update is sent to the host PC through Ethercontrol.

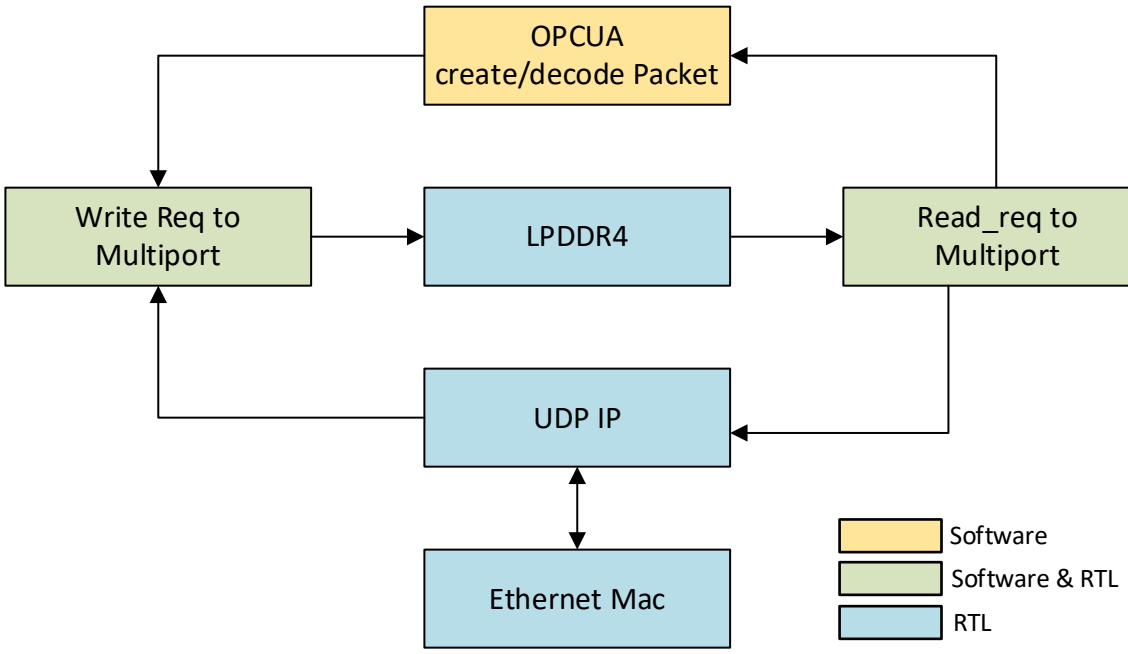

Figure 2.3. System Level User Flow

2.2.1.1. Memory Map

The Main System memory map is defined in Table 2.1.

Table 2.1. Main System Memory Map

Base Address	End Address	Range (Bytes)	Range (Bytes in hex)	Size (kB)	Block
F2000000	F20FFFFF	1048576	100000	1024	CPU CLINT
FC000000	FC3FFFFF	4194304	400000	4096	CPU PLIC TIMER
10000000	10000FFF	4096	1000	4	GPIO
F0000000	F00003FF	1024	400	1	RESERVED
F0000400	F1FFFFFF	32505856	1F00000	31744	RESERVED
F2100000	FBFFFFFF	1.66E+08	9F00000	162816	RESERVED
FC400000	FFFFFFFF	6.2E+07	3C00000	61440	RESERVED
100A0000	100A0FFF	4096	1000	4	CNN Co-Processor Unit (CCU)
00080000	000FFFFF	524288	80000	512	CPU Data Ram Port S0: base address 0x000C0000 Port S1: base address: 0x000E0000
00100000	0010FFFF	65536	10000	64	ISR RAM
10100000	10107FFF	32768	8000	32	FIFO DMA
10108000	0010FFFF	32768	8000	32	Ether Control
00000000	0007FFFF	524288	80000	512	SPI FLASH CONTROLLER
10090000	10091FFF	4096	1000	4	UART
10001000	10001FFF	4096	1000	4	SGMII TSE MAC Wrapper
10002000	10002FFF	4096	1000	4	UDP Stack
10110000	10117FFF	32768	8000	32	Multiport Extension (AXI4)
10091000	10091FFF	4096	1000	4	Multiport Extension (APB)

Note: Cache range of CPU lies from 0x00000000 to 0x0FFFFFFF.

2.2.2. Node System

The Node System architecture is shown in Figure 2.4. It consists of one AHBL Interconnect with three managers and eight subordinates.

The following are the Node System managers:

- RISC-V CPU Instruction Cache
- RISC-V CPU Data Cache
- FIFO DMA

The following are the Node System subordinates:

- ISR RAM
- Data RAM (port S0 and S1)
- Motor Control and PDM Data Collector (port S0 and S1)
- FIFO DMA
- EtherControl
- QSPI Memory Controller with perfect buffer (SPI Flash Controller)
- AHBL2APB bridge.
- SPI Manager
- I2C Manager

AHBL2APB bridge is connected to APB Interconnect which is having 3 APB interface-based subordinates SPI Manager, I2C Manager, and UART to interface different peripherals in the system (for example, sensors).

For Data RAM with two AHBL subordinate ports, see the description in the previous section. For Motor Control and PDM Data Collector, it has two AHBL subordinate ports (S0 and S1). Port S0 is used to access the Motor Control and PDM registers while port S1 is used to access the data collected by PDM Data Collector.

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

- During boot, the bootloader copies the ISR code from the external flash to internal ISR RAM. It then sets up the ISR function pointer to this internal memory address.
- The ISR code is integrated in the bitstream and firms the ISR code in the ISR RAM as ROM code.
- The first option can be used during initial development for debugging. The second option can be used in the final production release since it does not increase any system boot time.

The system is working at frequency of 75 MHz while the protocol is working at 125 MHz.

The CPU can access data from the Data RAM, access the register file inside EtherControl, and control the registers at FIFO DMA and QSPI Memory Controller. Either RISC-V CPU or FIFO DMA can move the data stored at the register file inside EtherControl to Motor Control block. They can also move the data collected by PDM Data Collector back to EtherControl and send out through the Ethernet upstream port.

There is one feature added in EtherControl IP in protocol layer. It supports additional frame/packet type 10 which enables the system to enhance performance while fetching bulk data. More details are given in the EtherControl user guide.

There are no major changes in EtherControl Node module.

Note: In Node System, previous version (AHBL subordinate support) of FIFO DMA, SPI Flash Controller, and Ethercontrol IP are used.

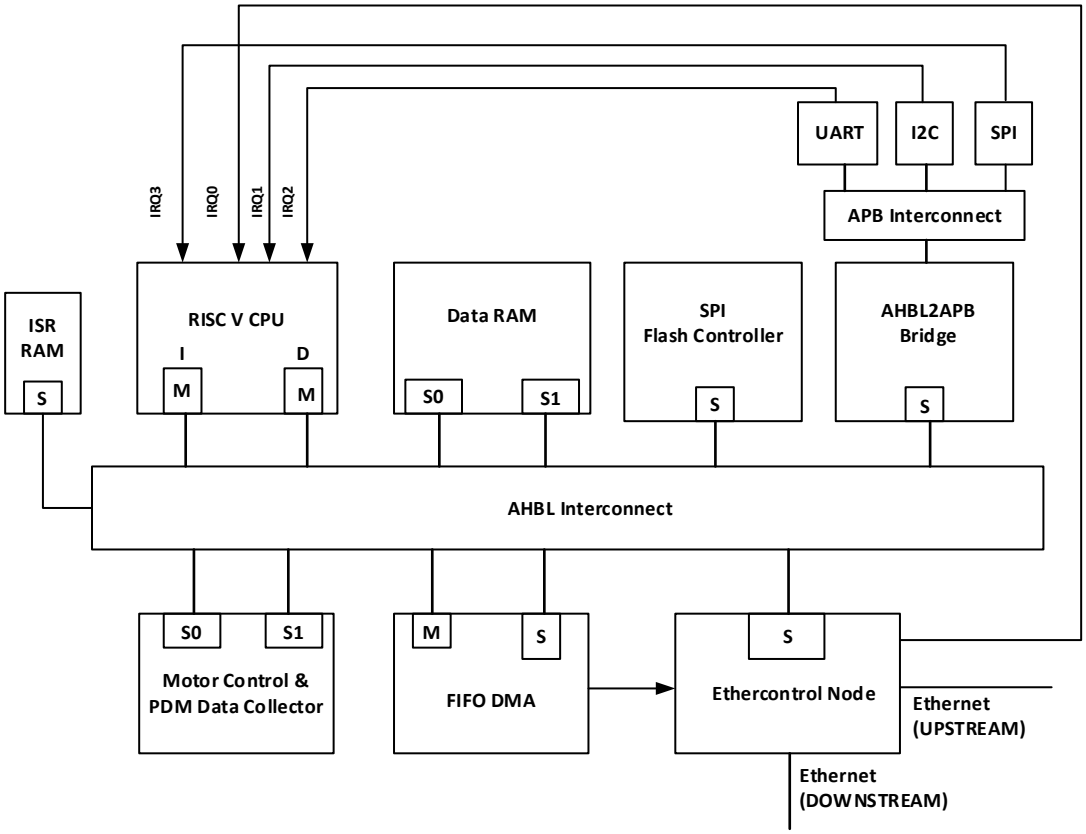

Figure 2.4. Node System Architecture

2.2.2.1. Node System Memory Map of Node System

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
00190000	00197FFF	32768	8000	32	CPU instruction RAM
00080000	000807FF	2048	800	2	CPU PIC TIMER
00080800	000BFFFF	260096	3F800	254	RESERVED
000C0000	000FFFFF	262144	40000	256	CPU Data Ram Port S0 base address: 0x000C0000 Port S1 base address: 0x000E0000
00100000	00107FFF	32768	8000	32	FIFO DMA
00108000	0010FFFF	32768	8000	32	EtherControl
00110000	0017FFFF	458752	70000	448	RESERVED
00000000	0007FFFF	512000	7D000	512	SPI Flash Controller
001864000	001867FF	1024	400	2	UART
00184000	00185FFF	8192	2000	8	Motor Control & PDM Data Collector Port S0 base address: 0x00184000 Port S1 base address: 0x00185000
00186000	00FFFFFF	15179776	E7A000	14824	RESERVED
01400000	01FFFFFF	16777216	1000000	16384	External SPI flash
001868000	00186BFF	1024	400	1	SPI Manager

Base Address	End Address	Range (Bytes)	Range (Bytes in hex)	Size (Kbytes)	Block
00186000	001863FF	1024	400	1	I^2C Manager

2.3. Ether Control IP

The Ether Control block is needed in both Main System and Node System. There is a Verilog parameter SYSTEM_TYPE, which sets this block as either Main System or Node System. For the Main System, there is no Ethernet Upstream; only two Ethernet downstream ports. For Node System, it has both Ethernet Upstream (1) and Ethernet Downstream(1) ports. For the last Node System, the Ethernet Downstream port can be disabled. Input/Output FIFO interface is selected using SYSTEM_TYPE parameter.

In the main system, the Ether Control IP has an output FIFO interface to send bulk data to DMA FIFO block while in the node system, the Ether Control block has an input FIFO interface to receive bulk data from DMA FIFO module, which is coming from the Data Collector IP. The AXI4 interface is used to support one host along with FIFO interface for bulk data.

The Sync Pulse generator block is available in the Ether Control manager only. It is used to generate pulse for the dynamic synchronization of nodes.

The Ether Control consists of an existing IP block (lscc_sgmii_gbe_pcs), Ether Connect, register file, and glue logic as shown in Figure 2.5.

Note: In Automate 3.0, PCIe based path is disabled by default.

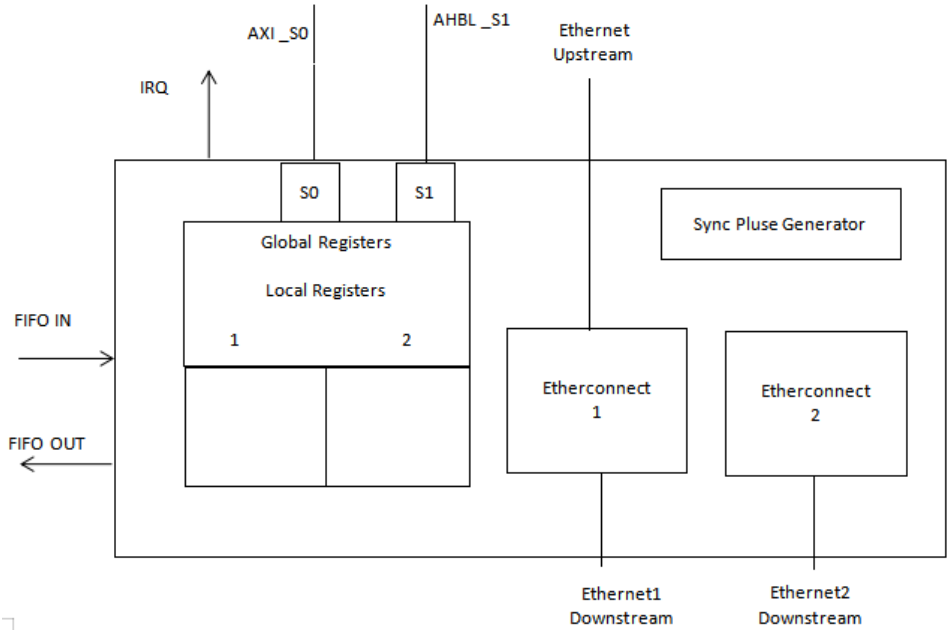

Figure 2.5. Ether Control Block Diagram

Table 2.3. Ether Control Interfaces

Interface	Direction	Description
IRQ	Output	Interrupt to RISC-V CPU
FIFO	Output	FIFO output to FIFO DMA
FIFO	Input	FIFO input to Ether Control
AXI4 Subordinate 0	Input and Output	AXI4 subordinate port for host 1 along with FIFO output interface to control IP.
AHBL Subordinate 1	Input and Output	AHBL subordinate port for host 2 independent of FIFO.

Interface	Direction	Description
Ethernet Upstream	Input and Output	Send Ethernet packets to Main System or Upper Node System This interface is disabled for Main System Ether Control.
Ethernet1 Downstream	Input and Output	Send ethernet packets to lower Node Systems This interface is disabled for the last Node System Ether Control.
Ethernet2 Downstream	Input and Output	Send ethernet packets to lower Node Systems This interface is disabled for the last Node System Ether Control.

2.3.1. Features

The key features of the Ether Control IP include:

- Real time Ethernet network support
- Two chain support (only one chain enabled in 3.0 system)
- Full Duplex data communication support
- RGMII interface support
- SGMII interface support
- AXI4 Node interfaces for controlling IP from AXI4 based manager block
- FIFO Interface for bulk data transfer (both normal and extended mode) (Only for AXI4 S0)
- Runtime Cable Break Detection Support
- Propagation Delay adjustment (Synchronization) Support
- Parameter based Main and Node Selection
- Maximum of 32 nodes support
- One AXI4 and One AHBL Bus support for EtherControl Main
- Max 256 bytes data length support
- Random Node access support – EtherControl Main
- RGMII/SGMII Selection
- 1G (125 MHz) physical interface support- RGMII/SGMII PHY, SFP support
- Dynamic/Runtime node scanning
- 4 kB Rx and Tx data buffers support
- Configuration Write (Motor Configuration), Status Read(Motor Status), Bulk data read(PDM data) – Normal and Extended
- Interrupt support (only for AXI4 S0)

2.3.2. EtherControl Main

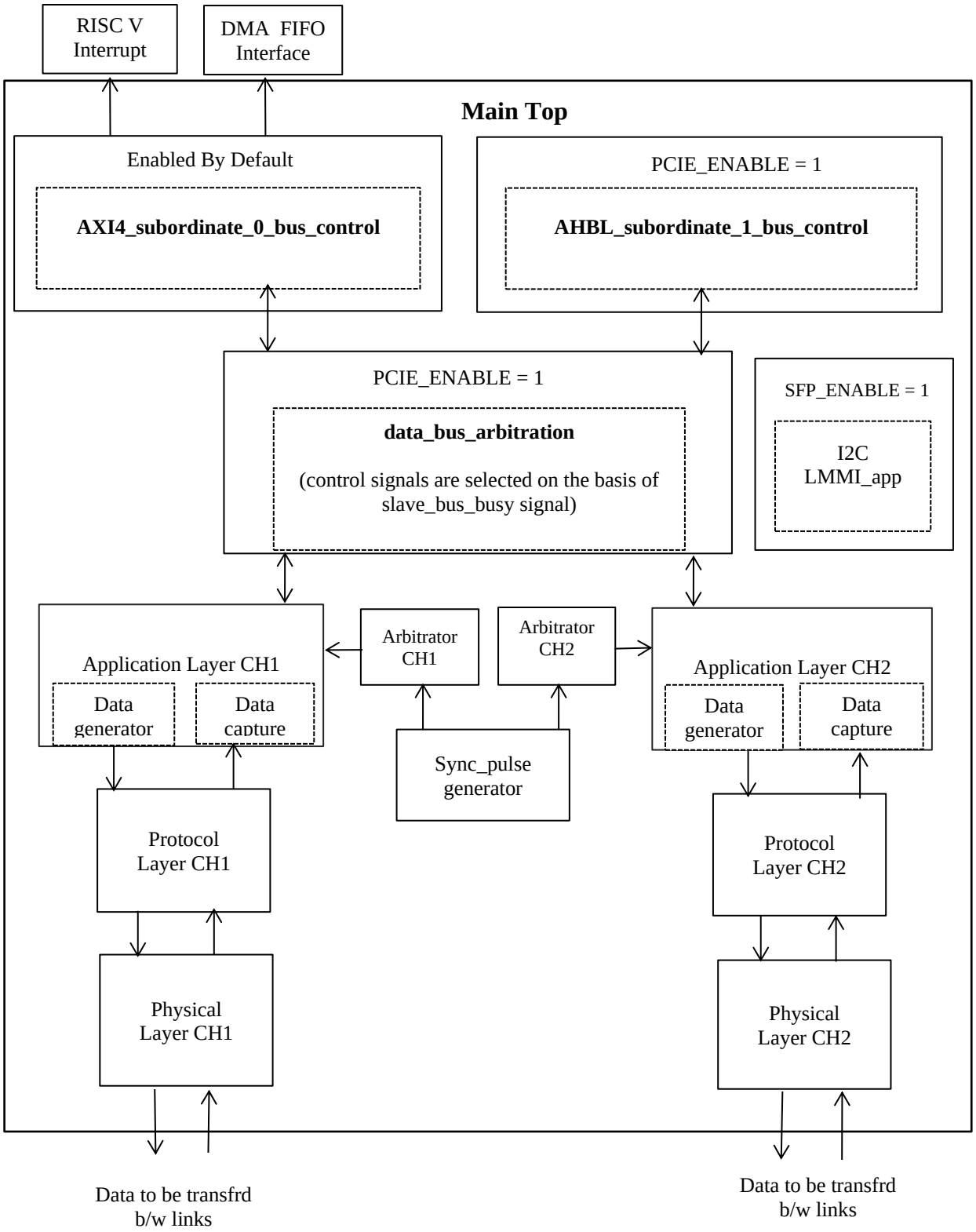

Figure 2.6. EtherControl Main Block Diagram

2.3.3. Register Description

2.3.3.1. EtherControl Main (RISC-V)

The register address map for AXI4 Bus 0 (RISC-V) shown in Table 2.4 specifies the available IP Core registers for the main system configuration. The offset of each register increments by four to allow easy interfacing with the processor and System Buses. In this case, each register is 32-bit wide wherein the used and unused bits are mentioned. The unused bits are treated as reserved – read access returns 0. The registers are divided into Global and Local ones. The global registers are common for all the chains, while the local ones are local to the respective chains.

Table 2.4. EtherControl Main Global Register Map (RISC-V)

EtherControl Register Name	Register Function	Base Address (0x10108000)	Access
DMACTR_R	DMA FIFO Enable/AXI4 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 starts for all chains	Base + 0x18	Read/Write
IP_STATUS_R	IP Busy Status	Base + 0x20	Read/Write
AXI4_TOUT_R	AXI4 Bus Timeout Count Register	Base + 0x28	Write

Table 2.5. EtherControl Main Local Chain 1 Register Map (RISC-V)

EtherControl Register Name	Register Function	Base Address (0x1010800)	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
MTRST_R_1	Node Motor Status Register	Base + 0x100 to 0x1FC	Read
DLY_R_1	Node Delay Register	Base + 0x200 to 0x2FC	Read

Table 2.6. EtherControl Main Local Chain 2 Register Map (RISC-V)

EtherControl Register Name	Register Function	Base Address (0x10108400)	Access
TXSTR_R_2	Start Transaction Register	Base + 0x00	Read/Write
PKTHD_R_2	Packet Head Register	Base + 0x04	Read/Write
FRNUM_R_2	Frame Number Register	Base + 0x08	Read/Write
NDCNT_R_2	Number of Node Register	Base + 0x0C	Read/Write
NDLN_R_2	Node Data Length Register	Base + 0x10	Read/Write
MTDT_R_2	Node Request Data Burst Register	Base + 0x14	Read/Write
RQDT_R_2	Node Request Type Register	Base + 0x18	Read/Write
RQAD_R_2	Node Address Register	Base + 0x1C	Read/Write
CRCNT_R_2	CRC Count Register	Base + 0x20	Read
INTR_R_2	Interrupt Info Register	Base + 0x24	Read
FSRREQD_R_2	FIFO Status Register Request Data	Base + 0x28	Read

EtherControl Register Name	Register Function	Base Address (0x10108400)	Access
MTRST_R_2	Node Motor Status Register	Base + 0x100 to 0x1FC	Read
DLY_R_2	Node Delay Register	Base + 0x200 to 0x2FC	Read

The Global register description is given below:

Table 2.7. DMA FIFO Enable/AXI4 Disable Register

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

DMACTR_R[0]: 0: DMA FIFO enabled, AXI4 disabled | 1: DMA FIFO disabled, AXI4 enabled

Table 2.8. PHY Link Status Register

PHLNK_R				Base + 0x04
Byte	3	2	1	0
Name	Physical Link Chain 2		Physical Chain 1	
Default	0	0	0	0
Access	R			

PHLNK_R[0]: 1: Main System PHY link established for chain 1 and 0: Main System PHY link not established for chain 1

PHLNK_R[15:1]: Each bit from bit 1 to bit 15 shows the link status of the respective nodes in chain 1

PHLNK_R[16]: 1: Main System PHY link established for chain 2 and 0: Main System PHY link not established for chain 2

PHLNK_R[31:17]: Each bit from bit 17 to bit 31 shows the link status of the respective nodes in chain 2

Table 2.9. Active Nodes Register

NDACT_R				Base + 0x08
Byte	3	2	1	0
Name	Active Node Chain 1+2	Active Node Chain 1+2	Active Node Chain 2	Active Node Chain 1
Default	0	0	0	0
Access	R			

NDACT_R[7:0]: Gives number of nodes actually connected physically to the system

NDACT_R[15:8]: Gives number of nodes actually connected physically to the system in chain 2

NDACT_R[31:16]: Gives total number of physically connected nodes in both chains

Table 2.10. FIFO Status Register for PDM Data

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

FSRPDM_R[0]: Empty signal of RX FIFO

FSRPDM_R[1]: Full signal of RX FIFO

FSRPDM_R[2]: Overflow error of RX FIFO

FSRPDM_R[3]: Underflow error of RX FIFO

FSRPDM_R[4]: Reserved

FSRPDM_R[5]: Reserved

FSRPDM_R[6]: Reserved

FSRPDM_R[7]: Reserved

Table 2.11. Clear Interrupt Received Register

CLRCVD_R				Base + 0x14
Byte	3	2	1	0
Name	CLRCVD_R			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

CLRCVD_R[0]: Received clr bit from CPU

CLRCVD_R[7:1]: Reserved

CLRCVD_R[31:8]: Reserved

Table 2.12. Interrupt Polling Register

ETHINTR_R				Base + 0x10
Byte	3	2	1	0
Name	Ethernet Interrupt from Chain 2		Ethernet Interrupt from Chain 1	
Default	Reserved	0	Reserved	0
Access	R			

ETHINTR_R[0]: Interrupt bit from Chain 1

ETHINTR_R[7:1]: Reserved

ETHINTR_R[15:8]: Reserved

ETHINTR_R[16]: Interrupt bit from Chain 2

ETHINTR_R[31:17]: Reserved

Table 2.13. Start Transaction in All Chains

TX_ALL_STRT_R				Base + 0x18
Byte	3	2	1	0
Name	TX_ALL_STRT_R			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

TX_ALL_START_R[0]: Received clear bit from CPU

TX_ALL_START_R[7:1]: Reserved

Table 2.14. IP Busy Register

IP_BUSY_R				Base + 0x20
Byte	3	2	1	0
Name	IP_BUSY_R			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

IP_BUSY_R[0]: 1 : AXI Bus 0 Busy | 0: AXI 0 bus Free (Only for reading)

IP_BUSY_R[1]: 1 : AXI Bus 1 Busy | 0: AXI 1 bus Free

IP_BUSY_R[7:2]: Reserved

Table 2.15. AXI4_TOUT_R

AXI4_TOUT_R				Base + 0x28
Byte	3	2	1	0
Name	AXI4_TOUT_R			
Default	0	0	0	0
Access	W			

AXI4_TOUT_R[31:0]: Sets the value of AXI4 timeout count to free the bus

The local register 1 description is given below:

Table 2.16. Chain 1 Start Transaction Register

TXSTR_R_1				Base + 0x00
Byte	3	2	1	0
Name	TXSTR_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

TXSTR_R_1[0]: 1: Start the transaction | 0: No transaction

Table 2.17. Chain 1 Packet Head Register

PKTHD_R_1				Base + 0x04
Byte	3	2	1	0
Name	PKTHD_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

PKTHD_R_1[0]: 1: User values are updated | 0: No update

Table 2.18. Chain 1 Frame Number Register

FRNUM_R_1				Base + 0x08
Byte	3	2	1	0
Name	FRNUM_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

FRNUM_R_1[7:0]: Frame number for the current frame

Table 2.19. Chain 1 Number of Node Register

NDCNT_R_1				Base + 0x0C
Byte	3	2	1	0
Name	NDCNT_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

NDCNT_R_1[7:0]: Number of nodes configured by the user

Table 2.20. Chain 1 Node Data Length Register

NDLN_R_1				Base + 0x10
Byte	3	2	1	0
Name	NDLN_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

NDLN_R_1[7:0]: Data length of nodes to be configured by the user

Table 2.21. Chain 1 Node Request Data Burst Register

MTDT_R_1				Base + 0x14
Byte	3	2	1	0
Name	MTDT_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

MTDT_R_1[7:0]: Data to be sent from the Main System to Node Systems by the user

Table 2.22. Chain 1 Node Request Type Register

RQDT_R_1				Base + 0x18
Byte	3	2	1	0
Name	RQDT_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

RQDT_R_1[7:0]: Type of data requested by the user

Table 2.23: Chain 1 Node Address Register

RQAD_R_1				Base + 0x1C
Byte	3	2	1	0
Name	RQAD_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

RQAD_R_1[7:0]: Address requested by the user

Table 2.24. Chain 1 CRC Count Register

CRCNT_R_1				Base + 0x20
Byte	3	2	1	0
Name	CRCNT_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R			

CRCNT_R_1[7:0]: Gives the count of errors generated by doing CRC on the data

Table 2.25. Chain 1 Interrupt Info Register

INTR_R_1				Base + 0x24
Byte	3	2	1	0
Name	INTR_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R			
RQAD_R_1				**Base + 0x1C**
Byte	3	2	1	0
Name	RQAD_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

INTR_R_1[31:0]: Gives the type of interrupt generated according to type of available data

0x01: Motor Configuration

0x02: Motor Status

0x03: PDM Data

0x04: Training Pkt

0x05: Pkt Head

0x06: Extended PDM Data

Table 2.26. Chain 1 FIFO Status Register Request Data

FSRREQD_R_1				Base + 0x28
Byte	3	2	1	0
Name	FSRREQD_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R			

FSRREQD_R_1[0]: Overflow error of TX 1 FIFO

FSRREQD_R_1[1]: Underflow error of TX 1 FIFO

FSRREQD_R_1[2]: Empty signal of TX 1 FIFO

FSRREQD_R_1[3]: Full signal of TX 1 FIFO

FSRREQD_R_1[4]: Reserved

FSRREQD_R_1[5]: Reserved

FSRREQD_R_1[6]: Reserved

FSRREQD_R_1[7]: Reserved

Table 2.27. Chain 1 Node Motor Status Register

MTRST_R_1				Base + 0x100 - 0x1FC
Byte	3	2	1	0
Name	MTRST_R_1			
Default	0	0	0	0
Access	R			

Base + 0x100: Node 1 status

Base + 0x104: Node 2 status (will progress like this for other nodes)

Table 2.28. Chain 1 Node Delay Register

DLY_1				Base +0x200 - 0x2FC
Byte	3	2	1	0
Name	DLY_1			
Default	0	0	0	0
Access	R/W			

Base + 0x200: Node 1 Delay

Base + 0x204: Node 2 Delay (will progress like this for other nodes)

The local register 2 descriptions are given below:

Table 2.29. Chain 2 Start Transaction Register

TXSTR_R_2				Base +0x00
Byte	3	2	1	0
Name	TXSTR_R_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

TXSTR_R_2[0]: 1: Start the transaction | 0: No transaction

Table 2.30. Chain 2 Packet Head Register

PKTHD_R_2				Base +0x04
Byte	3	2	1	0
Name	PKTHD_R_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

PKTHD_R_2[0]: 1: User values are updated | 0: No update

Table 2.31. Chain 2 Frame Number Register

FRNUM_R_2				Base +0x08
Byte	3	2	1	0
Name	FRNUM_R_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

FRNUM_R_2[7:0]: Frame number for the current frame

Table 2.32. Chain 2 Number of Node Register

NDCNT_R_2				Base +0x0C
Byte	3	2	1	0
Name	NDCNT_R_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

NDCNT_R_2[7:0]: Number of nodes configured by the user

Table 2.33. Chain 2 Node Data Length Register

NDLN_R_2				Base +0x10
Byte	3	2	1	0
Name	NDLN_R_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

NDLN_R_2[7:0]: Data length of nodes to be configured by the user

Table 2.34. Chain 2 Node Request Data Burst Register

MTDT_R_2				Base +0x14
Byte	3	2	1	0
Name	MTDT_R_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

MTDT_R_2[7:0]: Data to be sent from the Main System to Node Systems by the user

Table 2.35. Chain 2 Node Request Type Register

RQDT_R_2				Base +0x18
Byte	3	2	1	0
Name	RQDT_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

RQDT_R_2[7:0]: Type of data requested by the user

Table 2.36. Chain 2 Node Address Register

RQAD_R_2				Base +0x1C
Byte	3	2	1	0
Name	RQAD_R_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

RQAD_R_2[7:0]: Address requested by the user

Table 2.37: Chain 2 CRC Count Register

CRCNT_R_2				Base +0x20
Byte	3	2	1	0
Name	CRCNT_R_2			
Default	0	0	0	0
Access	R			

CRCNT_R_2[7:0]: Gives the count of errors generated by doing CRC on the data

Table 2.38: Interrupt Info Register

INTR_R_2				Base +0x24
Byte	3	2	1	0
Name	INTR_R_2			
Default	0	0	0	0
Access	R			

INTR_R_2[31:0]: Gives the type of interrupt generated according to type of available data

0x01: Motor Configuration

0x02: Motor Status

0x03: PDM Data

0x04: Training Pkt

0x05: Pkt Head

0x06: Extended PDM Data

Table 2.39: Chain 2 FIFO Status Register Request Data

FSRREQD_R_2				Base +0x28
Byte	3	2	1	0
Name	FSRREQD_R_2			
Default	Reserved	Reserved	Reserved	0
Access	R			

FSRREQD_R_2[0]: Overflow error of TX 1 FIFO

FSRREQD_R_2[1]: Underflow error of TX 1 FIFO

FSRREQD_R_2[2]: Empty signal of TX 1 FIFO

FSRREQD_R_2[3]: Full signal of TX 1 FIFO

FSRREQD_R_2[4]: Reserved

FSRREQD_R_2[5]: Reserved
FSRREQD_R_2[6]: Reserved
FSRREQD_R_2[7]: Reserved

Table 2.40: Chain 2 Node Motor Status Register

MTRST_R_2				Base +0x100 – 0x1FC
Byte	3	2	1	0
Name	MTRST_R_2			
Default	0	0	0	0
Access	R			

Base + 0x100: Node 1 status
Base + 0x104: Node 2 status (will progress like this for other nodes)

Table 2.41: Chain 2 Node Delay Register

DLY_R_2				Base +0x200 – 0x2FC
Byte	3	2	1	0
Name	DLY_R_2			
Default	0	0	0	0
Access	R			

Base + 0x200: Node 1 Delay
Base + 0x204: Node 2 Delay ……..(will progress like this for other nodes)

2.3.3.2. EtherControl Main (PCIe)

The register address map for AHBL Bus 1 (PCIe) shown in Table 2.42 specifies the available IP Core registers for main system configuration. The offset of each register increments by eight to allow easy interfacing with the Processor and System Buses. In this case, each register is 64-bit wide wherein the used and unused bits are mentioned. The unused bits are treated as reserved – read access returns 0. The registers are divided into Global and Local ones. The global registers are common for all the chains while the local ones are local to the respective chains.

Table 2.42: EtherControl Main Global Register Map (PCIe)

PCIe Register Name	Register Function	Base Address (0x10118000)	Access	Used Bits
DMACTR_P	DMA FIFO Enable/AHBL Disable Register	Base + 0x00	Read/Write	[31:0]
PHLNK_P	Phy Link Status Register	Base + 0x00	Read	[63:32]
NDACT_P	Active Nodes Register	Base + 0x08	Read	[31:0]
FSRPDM_P	FIFO Status Register for PDM Data CDC	Base + 0x08	Read	[63:32]
ETHINTR_P	Interrupt Poll Register	Base + 0x10	Read	[31:0]
CLRCVD_P	Clear Interrupt Received Register	Base + 0x10	Read/Write	[63:32]
TX_ALL_STRT_P	Transaction starts for all chains	Base + 0x18	Read/Write	[31:0]
IP_STATUS_P	IP Busy Status	Base + 0x20	Read/Write	[31:0]
AHBL_TOUT_P	AHBL Bus Timeout Count Register	Base + 0x28	Write	[31:0]
DTOUT_P	Node Response PDM Data Register	0x00108400 + 0x40	Read	[63:0]

Note: For PCIe based path, the PDM data goes through the AHBL interface to the DMACTR register that needs to be controlled.

Table 2.43. EtherControl Main Local Chain 1 Register Map (PCIe)

PCIe Register Name	Register Function	Base Address (0x10118100)	Access	Used Bits
TXSTR_P_1	Start Transaction Register	Base + 0x00	Read/Write	[31:0]
PKTHD_P_1	Packet Head Register	Base + 0x00	Read/Write	[63:32]
FRNUM_P_1	Frame Number Register	Base + 0x08	Read/Write	[31:0]
NDCNT_P_1	Number of Node Register	Base + 0x08	Read/Write	[63:32]
NDLN_P_1	Node Data Length Register	Base + 0x10	Read/Write	[31:0]
FSRREQD_P_1	FIFO Status Register Request Data	Base + 0x10	Read	[63:32]
RQDT_P_1	Node Request Type Register	Base + 0x18	Read/Write	[31:0]
RQAD_P_1	Node Address Register	Base + 0x18	Read/Write	[63:32]
CRCNT_P_1	CRC Count Register	Base + 0x20	Read	[31:0]
INTR_P_1	Interrupt Info Register	Base + 0x20	Read	[63:32]
MTDT_P_1	Node Request Data Burst Register	Base + 0x28	Read/Write	[63:0]
MTRST_P_1	Node Motor Status Register	Base + 0x100 to 0x1FC	Read	[63:0]
DLY_P_1	Node Delay Register	Base + 0x200 to 0x2FC	Read	[63:0]

Table 2.44. EtherControl Main Local Chain 2 Register Map (PCIe)

PCIe Register Name	Register Function	Base Address (0x10118400)	Access	Used Bits
TXSTR_P_2	Start Transaction Register	Base + 0x00	Read/Write	[31:0]
PKTHD_P_2	Packet Head Register	Base + 0x00	Read/Write	[63:32]
FRNUM_P_2	Frame Number Register	Base + 0x08	Read/Write	[31:0]
NDCNT_P_2	Number of Node Register	Base + 0x08	Read/Write	[63:32]
NDLN_P_2	Node Data Length Register	Base + 0x10	Read/Write	[31:0]
FSRREQD_P_2	FIFO Status Register Request Data	Base + 0x10	Read/Write	[63:32]
RQDT_P_2	Node Request Type Register	Base + 0x18	Read/Write	[31:0]
RQAD_P_2	Node Address Register	Base + 0x18	Read/Write	[63:32]

PCIe Register Name	Register Function	Base Address (0x10118400)	Access	Used Bits
CRCNT_P_2	CRC Count Register	Base + 0x20	Read	]31:0]
INTR_P_2	Interrupt Info Register	Base + 0x20	Read	[63:32]
MTDT_P_2	Node Request Data Burst Register	Base + 0x28	Read	[63:0]
MTRST_P_2	Node Motor Status Register	Base + 0x100 to 0x1FC	Read	[63:0]
DLY_P_2	Node Delay Register	Base + 0x200 to 0x2FC	Read	[63:0]

The Global register description is given below:

Table 2.45. DMA FIFO Enable/AHBL Disable Register

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

DMACTR_P[0]: 0: DMA FIFO enabled, AHBL disabled | 1: DMA FIFO disabled, AHBL enabled

Table 2.46. PHY Link Status Register

PHLNK_P				Base + 0x00
Byte	7	6	5	4
Name	Physical Link Chain 2		Physical Chain 1	
Default	0	0	0	0
Access	R			

PHLNK_P[32]: 1: Main System PHY link established for chain 1 and 0: Main System PHY link not established for chain 1
PHLNK_P[47:33]: Each bit from bit 33 to bit 47 shows the link status of the respective nodes in chain 1
PHLNK_P[48]: 1: Main System PHY link established for chain 2 and 0: Main System PHY link not established for chain 2
PHLNK_P[63:49]: Each bit from bit 49to bit 63 shows the link status of the respective nodes in chain 2

Table 2.47. Active Nodes Register

NDACT_P				Base + 0x08
Byte	3	2	1	0
Name	Active Node Chain 1+2	Active Node Chain 1+2	Active Node Chain 2	Active Node Chain 1
Default	0	0	0	0
Access	R			

NDACT_P[7:0]: Gives number of nodes actually connected physically to the system

Table 2.48. FIFO Status Register for PDM Data

FSRPDM_P				Base + 0x08
Byte	7	6	5	4
Name	FSRPDM_P			
Default	Reserved	Reserved	Reserved	0
Access	R			

FSRPDM_P[32]: Empty signal of RX FIFO
FSRPDM_P[33]: Full signal of RX FIFO
FSRPDM_P[34]: Overflow error of RX FIFO
FSRPDM_P[35]: Underflow error of RX FIFO
FSRPDM_P[36]: Reserved
FSRPDM_P[37]: Reserved
FSRPDM_P[38]: Reserved
FSRPDM_P[39]: Reserved

Table 2.49. Interrupt Polling Register

ETHINTR_P				Base + 0x10
Byte	3	2	1	0
Name	Ethernet Interrupt from Chain 2		Ethernet Interrupt from Chain 1	
Default	Reserved	0	Reserved	0
Access	R/W			

ETHINTR_P[0]: Interrupt bit from Chain 1
ETHINTR[7:1]: Reserved
ETHINTR[15:8]: Reserved
ETHINTR[16]: Interrupt bit from Chain 2
ETHINTR[31:17]: Reserved

Table 2.50. Clear Interrupt Received Register

CLRCVD_P				Base + 0x10
Byte	7	6	5	4
Name	CLRCVD_P			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

CLRCVD[39]: Received CLR bit from CPU
CLRCVD[39:33]: Reserved

Table 2.51. Start Transaction in All Chains

TX_ALL_STRT_P				Base + 0x18
Byte	3	2	1	0
Name	TX_ALL_STRT_P			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

TX_ALL_STRT_P [0]: 1: Start the transaction | 0: No transaction

Table 2.52. IP Busy Register

IP_Busy_P				Base + 0x20
Byte	3	2	1	0
Name	AHBL_Bus_P			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

AHBL_BUSY_P[0]: 1: AHBL Bus 0 Busy | 0: AHBL 0 bus Free (only for reading)
AHBL_BUSY_P[1]: 1 : AHBL Bus 1 Busy | 0: AHBL 1 bus Free
AHBL_BUSY_P[7:2]: Reserved

Table 2.53. AHBL Bus Timeout Count Register

AHBL_TOUT_P				Base + 0x28
Byte	3	2	1	0
Name	AHBL_TOUT_P			
Default	Reserved	Reserved	Reserved	0
Access	W			

AHBL_TOUT_P[31:0]: Sets the value of AHBL timeout count to free the bus

Table 2.54. Node Response PDM Data Register

DTOUT_P								0x00108400 + 0x40
Byte	7	6	5	4	3	2	1	0
Name	DTOUT_P							
Default	0	0	0	0	0	0	0	0
	R							

DTOUT_P[63:0]: PDM Data out from the nodes which are requested by the user
The local register 1 description is given below:

Table 2.55. Chain 1 Start Transaction Register

TXSTR_P_1				Base + 0x00
Byte	3	2	1	0
Name	TXSTR_P_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

TXSTR_P_1[0]: 1: Start the transaction | 0: No transaction

Table 2.56. Chain 1 Packet Head Register

PKTHD_P_1				Base + 0x00
Byte	7	6	5	4
Name	PKTHD_P_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

PKTHD_P_1[32]: 1: User values are updated | 0: No update

Table 2.57. Chain 1 Frame Number Register

FRNUM_P_1				Base + 0x08
Byte	3	2	1	0
Name	FRNUM_P_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

FRNUM_P_1[7:0]: Frame number for the current frame

Table 2.58. Chain 1 Number of Node Register

NDCNT_P_1				Base + 0x08
Byte	7	6	5	4
Name	NDCNT_P_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

NDCNT_P_1[39:32]: Number of nodes configured by the user

Table 2.59. Chain 1 Node Data Length Register

NDLN_P_1				Base + 0x10
Byte	3	2	1	0
Name	NDLN_P_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

NDLN_1[7:0]: Data length of nodes to be configured by the user

Table 2.60. Chain 1 FIFO Status Register Request Data

FSRREQD_P_1				Base + 0x10
Byte	7	6	5	4
Name	FSRREQD_P_1			
Default	Reserved	Reserved	Reserved	0
Access	R			

FSRREQD_P_1[32]: Overflow error of TX 1 FIFO
FSRREQD_P_1[33]: Underflow error of TX 1 FIFO
FSRREQD_P_1[34]: Empty signal of TX 1 FIFO
FSRREQD_P_1[35]: Full signal of TX 1 FIFO
FSRREQD_P_1[36]: Reserved
FSRREQD_P_1[37]: Reserved
FSRREQD_P_1[38]: Reserved
FSRREQD_P_1[39]: Reserved

Table 2.61. Chain 1 Node Request Type Register

RQDT_P_1				Base + 0x18
Byte	3	2	1	0
Name	RQDT_P_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

RQDT_1[7:0]: Type of data requested by the user

Table 2.62. Chain 1 Node Address Register

RQAD_P_1				Base + 0x18
Byte	3	2	1	0
Name	RQAD_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

RQAD_P_1[39:32]: Address requested by the user

Table 2.63. Chain 1 CRC Count Register

CRCNT_P_1				Base + 0x20
Byte	3	2	1	0
Name	CRCNT_P_1			
Default	Reserved	Reserved	Reserved	0
Access	R			

CRCNT_P_1[7:0]: Gives the count of errors generated by doing CRC on the data

Table 2.64. Chain 1 Interrupt Info Register

INTR_P_1				Base + 0x24
Byte	7	6	5	4
Name	INTR_P_1			
Default	Reserved	Reserved	Reserved	0
Access	R			

INTR_P_1[63:32]: Gives the type of interrupt generated according to type of available data
0x01: Motor Configuration
0x02: Motor Status
0x03: PDM Data
0x04: Training Pkt
0x05: Pkt Head
0x06: Extended PDM Data

Table 2.65. Chain 1 FIFO Status Register Request Data

FSRREQD_1				Base + 0x28
Byte	3	2	1	0
Name	FSRREQD_1			
Default	Reserved	Reserved	Reserved	0
Access	R			

FSRREQD_1[0]: Overflow error of TX 1 FIFO
FSRREQD_1[1]: Underflow error of TX 1 FIFO
FSRREQD_1[2]: Empty signal of TX 1 FIFO
FSRREQD_1[3]: Full signal of TX 1 FIFO
FSRREQD_1[4]: Reserved
FSRREQD_1[5]: Reserved
FSRREQD_1[6]: Reserved
FSRREQD_1[7]: Reserved

Table 2.66. Chain 1 Node Request Burst Register

MTDT_P_1								Base + 0x28
Byte	7	6	5	4	3	2	1	0
Name	MTDT_P_1							
Default	0	0	0	0	0	0	0	0
	R/W							

MTDT_P_1[63:0]: Data to be sent from the main to nodes by the user

Table 2.67. Chain 1 Node Motor Status Register

MTRST_P_1								Base + 0x100 – 0x1FC
Byte	7	6	5	4	3	2	1	0
Name	MTRST_P_1							
Default	0	0	0	0	0	0	0	0
	R							

Base + 0x100: Node 1 status
Base + 0x104: Node 2 status (will progress like this for other nodes)

Table 2.68. Chain 1 Node Delay Register

DLY_P_1								Base + 0x200 – 0x1FC
Byte	7	6	5	4	3	2	1	0
Name	DLY_P_1							
Default	0	0	0	0	0	0	0	0
	R							

Base + 0x200: Node 1 Delay
Base + 0x204: Node 2 Delay (will progress like this for other nodes)
The local register 2 descriptions are given below:

Table 2.69. Chain 2 Start Transaction Register

TXSTR_P_2				Base +0x00
Byte	3	2	1	0
Name	TXSTR_P_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

TXSTR_P_2[0]: 1: Start the transaction | 0: No transaction

Table 2.70. Chain 2 Packet Head Register

PKTHD_P_2				Base +0x00
Byte	7	6	5	4
Name	PKTHD_P_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

PKTHD_P_2[32]: 1: User values are updated | 0: No update

Table 2.71. Chain 2 Frame Number Register

FRNUM_P_2				Base +0x08
Byte	3	2	1	0
Name	FRNUM_P_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

FRNUM_P_2[7:0]: Frame number for the current frame

Table 2.72. Chain 2 Number of Node Register

NDCNT_P_2				Base +0x08
Byte	7	6	5	4
Name	NDCNT_P_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

NDCNT_P_2[39:32]: Number of nodes configured by the user

Table 2.73. Chain 2 Node Data Length Register

NDLN_P_2				Base +0x10
Byte	3	2	1	0
Name	NDLN_P_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

NDLN_P_2[7:0]: Data length of nodes to be configured by the user

Table 2.74. Chain 2 FIFO Status Register Request Data

FSRREQD_P_2				Base +0x10
Byte	7	6	5	4
Name	FSRREQD_P_2			
Default	Reserved	Reserved	Reserved	0
Access	R			

FSRREQD_P_2[32]: Overflow error of TX 1 FIFO
FSRREQD_P_2[33]: Underflow error of TX 1 FIFO
FSRREQD_P_2[34]: Empty signal of TX 1 FIFO
FSRREQD_P_2[35]: Full signal of TX 1 FIFO
FSRREQD_P_2[36]: Reserved
FSRREQD_P_2[37]: Reserved
FSRREQD_P_2[38]: Reserved
FSRREQD_P_2[39]: Reserved

Table 2.75. Chain 2 Node Request Type Register

RQDT_P_2				Base + 0x14
Byte	3	2	1	0
Name	RQDT_P_1			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

RQDT_P_2[7:0]: Type of data requested by the user

Table 2.76. Chain 2 Node Address Register

RQAD_P_2				Base + 0x14
Byte	7	6	5	4
Name	RQAD_P_2			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

RQAD_P_2[39:32]: Address requested by the user

Table 2.77. Chain 2 CRC Count Register

CRCNT_P_2				Base + 0x20
Byte	3	2	1	0
Name	CRCNT_P_2			
Default	Reserved	Reserved	Reserved	0
Access	R			

CRCNT_P_2[7:0]: Gives the count of errors generated by doing CRC on the data

Table 2.78. Interrupt Info Register

INTR_P_2				Base + 0x20
Byte	7	6	5	4
Name	INTR_P_2			
Default	0	0	0	0
Access	R			

INTR_P_2[63:32]: Gives the type of interrupt generated according to type of available data
0x01: Motor Configuration
0x02: Motor Status
0x03: PDM Data
0x04: Training Pkt
0x05: Pkt Head
0x06: Extended PDM Data

Table 2.79. Chain 2 Node Request Burst Register

MTDT_P_2								Base + 0x28
Byte	7	6	5	4	3	2	1	0
Name	MTDT_P_1							
Default	0	0	0	0	0	0	0	0
Access	R/W							

MTDT_P_2[63:0]: Data to be sent from the main to nodes by the user

Table 2.80. Chain 2 Node Motor Status Register

MTRST_P_2								Base + 0x100 – 0x1FC
Byte	7	6	5	4	3	2	1	0
Name	MTRST_P_2							
Default	0	0	0	0	0	0	0	0
Access	R							

Base + 0x100: Node 1 status
Base + 0x104: Node 2 status (will progress like this for other nodes)

Table 2.81. Chain 2 Node Delay Register

DLY_P_2								Base + 0x200 – 0x2FC
Byte	7	6	5	4	3	2	1	0
Name	DLY_P_2							
Default	0	0	0	0	0	0	0	0
Access	R							

Base + 0x200: Node 1 status

Base + 0x204: Node 2 status (will progress like this for other nodes)

2.3.4. EtherControl Node

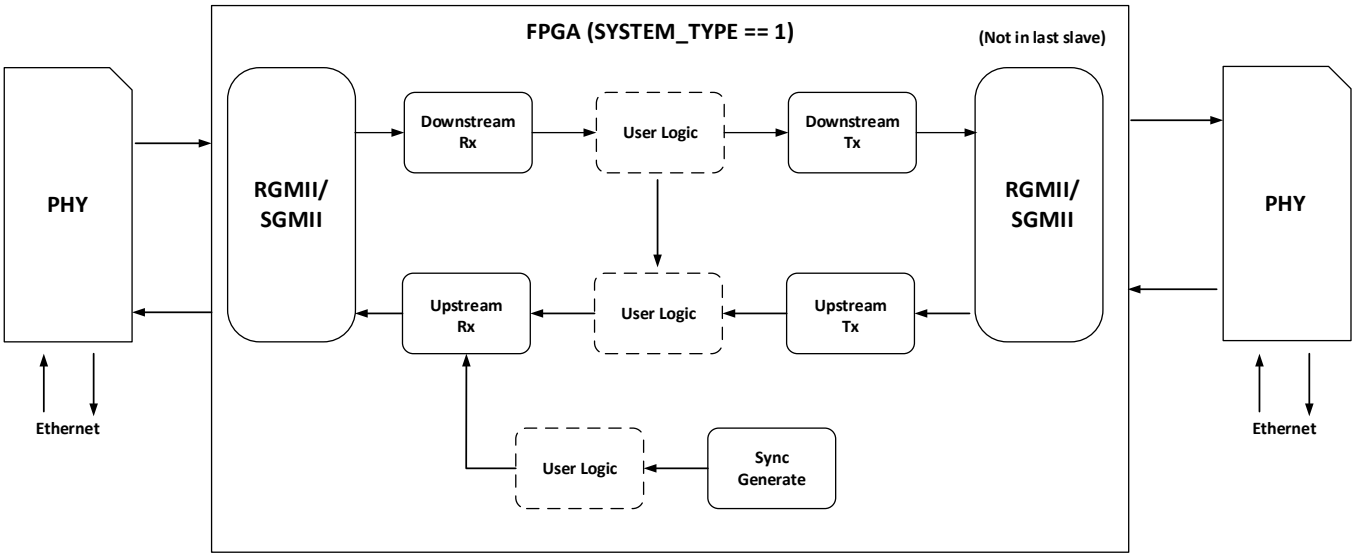

Figure 2.7. EtherControl Node

Table 2.82. EtherControl Node Register Map

EtherControl Register Name	Register Function	Address	Access
DMACTR	DMA control Register	Base + 0x00	Read/Write
FFDT	FIFO data Register	Base + 0x04	Write
RCMTR	Motor Status Register	Base + 0x08	Write
DMST	DMA Done Indication Register	Base + 0x0C	Write
INTST	Interrupt Status Register	Base + 0x10	Read
MTAD	Motor Config/Status Address Register or PDM Data Transfer Size Register	Base + 0x14	Read
MTDT	Motor Config Data Register	Base + 0x18	Read
FFER	FIFO error Register	Base + 0x1C	Read
CLRCVD	Clear Interrupt Received Register	Base + 0x3C	Read/Write

Table 2.83. DMA Control Register

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

DMACTR[0]: 0: DMA FIFO enabled, AHBL disabled | 1: DMA FIFO disabled, AHBL enabled

Table 2.84. FIFO Data Register

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

FFDT[7:0]: Data incoming to the FIFO present in EtherControl Node System

Table 2.85. Motor Status Register

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

RCMTR[7:0]: Motor status

Table 2.86. DMA Done Indication Register

DMST				Base +0x0C
Byte	3	2	1	0
Name	DMST			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

DMST[7:0]: DMA done status

Table 2.87. Interrupt Status Register

INTST				Base +0x10
Byte	3	2	1	0
Name	INTST			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

INTST[7:0]: Gives the type of interrupt thrown by the node system

Table 2.88. Motor Config/Status Address Register (or) PDM Data Transfer Size Register

MTAD				Base +0x14
Byte	3	2	1	0
Name	MTAD			
Default	0	0	0	0
Access	R			

MTAD[31:0]: Gives the config/status address or PDM data transfer size

Table 2.89. Motor Configuration Data Register

MCDR				Base +0x18
Byte	3	2	1	0
Name	MCDR			
Default	0	0	0	0
Access	R			

MCDR[31:0]: Gives the data available for motor configuration

Table 2.90. FIFO Error Register

FFER				Base +0x1C
Byte	3	2	1	0
Name	FFER			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

FFER[0]: Overflow error of FIFO
FFER[1]: Underflow error of FIFO
FFER[2]: Downstream sync signal for particular node
FFER[3]: Empty status of FIFO
FFER[4]: Full status of FIFO
FFER[5]: Reserved
FFER[6]: Reserved
FFER[7]: Reserved

Table 2.91. Clear Interrupt Received Register

CLRCVD				Base +0x3C
Byte	3	2	1	0
Name	CLRCVD			
Default	Reserved	Reserved	Reserved	0
Access	R/W			

CLRCVD[0]: Received clr bit from CPU
CLRCVD[7:1] : Reserved

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 EtherControl manager Bus 0. The other interface is active for node and has the pdm data from the motor control data collector block.

This block also has an AXI4 subordinate and manager interface. The register space for this block is as shown in Table 2.92.

The AXI4 Subordinate interface is used to control DMA operations by external manager (which is CPU) and AXI4 manager interface is used to perform for DMA operations. More information about this IP is given FIFO DMA user guide.

Table 2.92. 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 2.93. 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 the destination base address.
CNTR[2-7]: Reserved

Table 2.94. 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 2.95. 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 2.96. Write Status Register

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

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

Table 2.97. Read Status Register

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

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

2.5. SGMII TSE MAC Wrapper

The SGMII TSE MAC Wrapper block is needed in Main System to receive and transmit data from and to host and is developed using existing SGMII and TSE MAC IPs.

SGMII IP converts the serial data coming from host through ethernet cable connected to onboard SFP into GMII format (8-bit data) and transmits it to TSE MAC IP. I2C Manager and LMMI App IP are used for SFP Configuration and linkup.

TSEMAC IP core contains all necessary logic, interfacing, and clocking infrastructure necessary to integrate an external industry-standard Ethernet PHY with aninternal 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 TSEMACextracts different components of a frame and transfers them to higher applications through the FIFO interface.

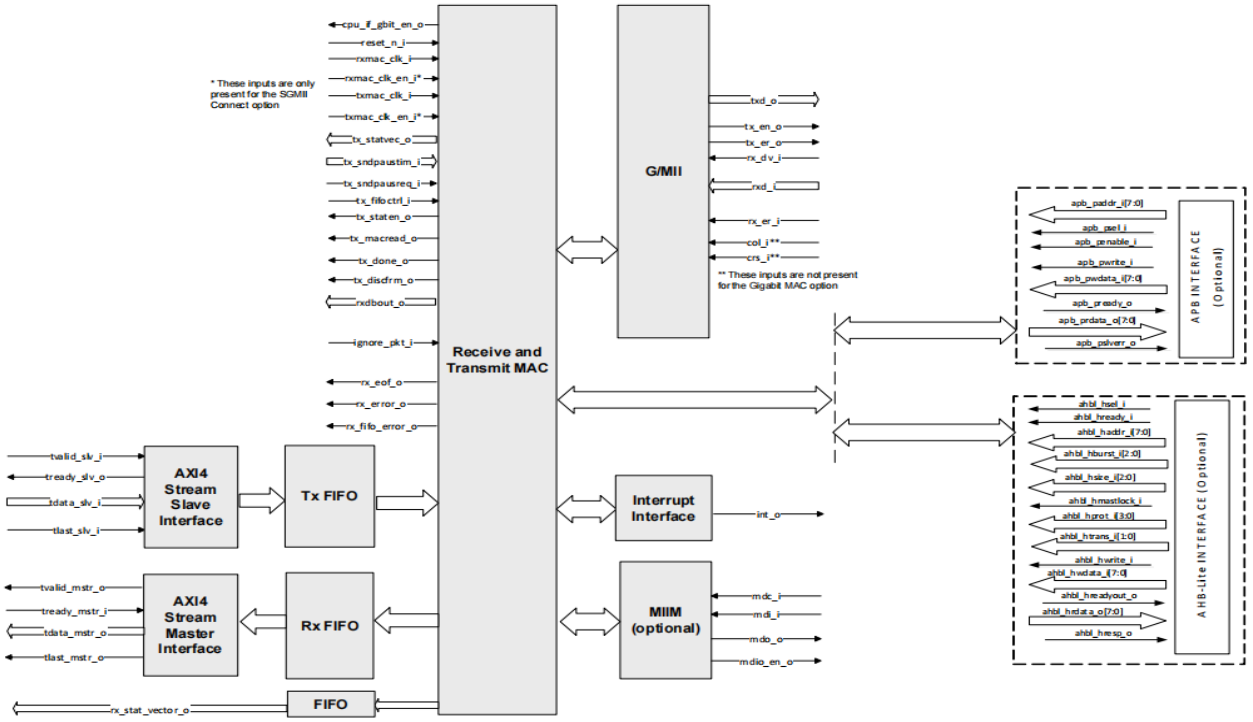

Figure 2.8. Top-Level Block Diagram of TSE_MAC IP (SGMII Easy Connect)

Table 2.98:Register Map of SGMII TSE MAC IP

	Register Name	Description
00H - 01H	Mode register	Enables/Disables IP Core functions.
02H - 03H	Transmit and Receive Control register	This register can be overwritten only when Rx MAC and Tx MAC are disabled. This register controls various features of MAC.
04H - 05H	Maximum Packet Size register	This register can be overwritten only when MAC is disabled. All frames longer than the value (number of bytes) in this register is tagged as long frames.
08H - 09H	Inter-Packet Gap register	Time between packet transmission.
0AH - 0BH	TSEMAC IP Core Address register 0	Contains the Ethernet address of the port.
0CH - 0DH	TSEMAC IP Core Address register 1	
0EH - 0FH	TSEMAC IP Core Address register 2	
12H - 13H	Transmit and Receive Status register	This register reports events that have occurred while receiving or transmitting a packet.
14H - 15H	GMII Management Interface Control register	The GMII Management Access register controls the Management Interface Module. This register can be overwritten only when the interface is not busy. A write operation is ignored when the interface is busy.
16H - 17H	GMII Management Data register	The contents of this register are transmitted when a Write operation is to be performed. When a Read operation is performed, this register contains the value that was read from a PHY register.
32H - 33H	VLAN Tag Length/type register	The VLAN tag register has the VLAN TAG field of the most recent tagged frame that was received. This is a read-only register.
22H - 23H	Multicast_table_0 register	Multicast Table. Eight tables that make a 64-bit hash.
24H - 25H	Multicast_table_1 register	
26H - 27H	Multicast_table_2 register	
28H - 29H	Multicast_table_3 register	

The main function of Rx MAC is to accept formatted data from G/MII interface and pass it to LMMI interface through FIFO. During this, Rx MAC performs following functions: detect the start of the frame, compare MAC address, re-calculate CRC, process the control frame, and pass it to the flow control module.

Transmit MAC (Tx MAC) is responsible for controlling access to the physical medium. Tx MAC reads data from a Tx FIFO when the FIFO is not empty and when it detects an active tx_fifoavail. Tx MAC then formats this data into an Ethernet packet and passes it to the G/MII module.

For more details on the IP, refer to Tri-Speed Ethernet MAC IP Core - Lattice Radiant Software (FPGA-IPUG-02084).

2.6. UDP Stack

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

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

For more details, refer to the IP User Guide.

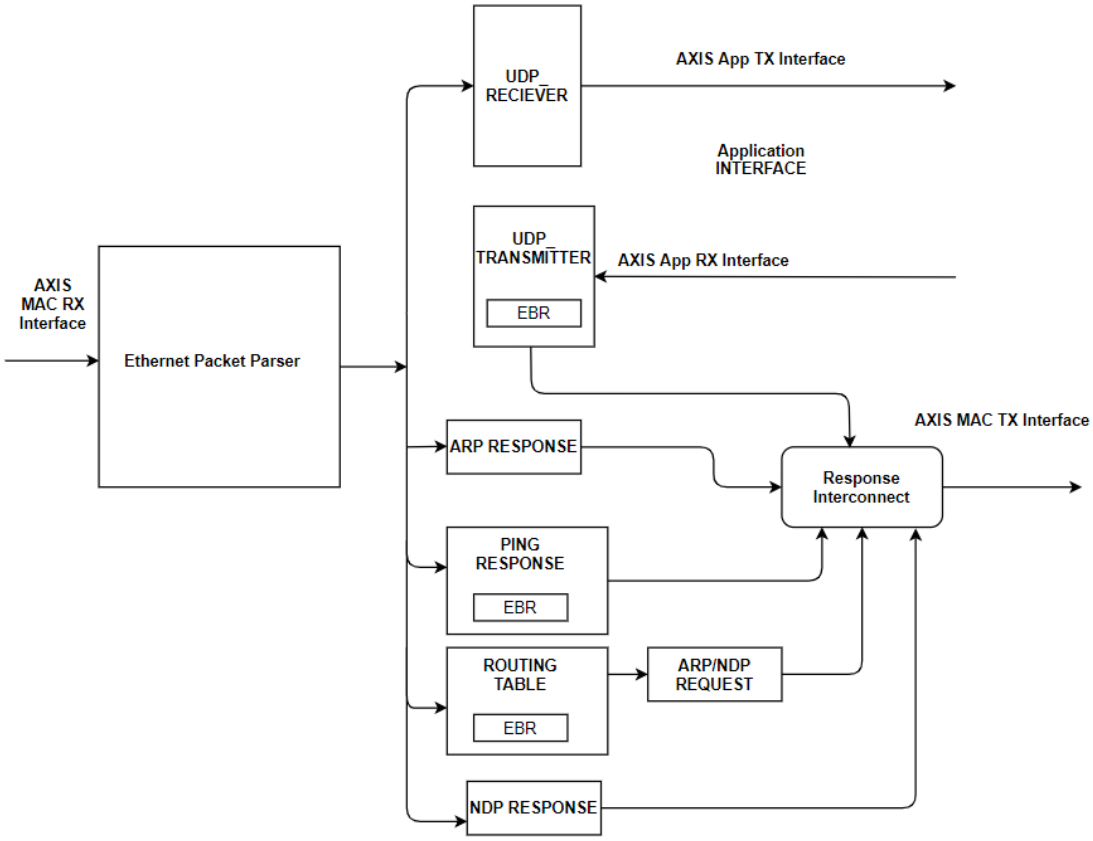

Figure 2.9: UDP Stack Top Level Architecture

The register Map of UDP Stack IP is given below:

Table 2.99. Register Map of UDP Stack

Register Name	Offset	Description
REG_ENABLE_IPV6	0x00	0x00: [0] reg_enable_ipv6. [31:1]: reserved
REG_MAC_ADD_L	0x04	0x04: reg_mac_add_l [31:0]
REG_MAC_ADD_H	0x08	0x08: [15:0] reg_mac_add_h [47:32] [31:16] reserved
REG_IPV4_ADD	0x0C	0x0C: reg_ipv4_add [31:0]
REG_GATEWAY_ADD	0x10	0x10: reg_gateway_add [31:0]
REG_SUBNET_MASK	0X14	0x14: reg_subnet_mask [31:0]
REG_RX_PORT	0x18	0x18: [15:0] reg_rx_port [31:16] reserved.
REG_TX_UDP_DST_SRC_PORT	0x1C	0x1C: [31:16]: reg_tx_udp_dst_port [15:0]: reg_tx_udp_src_port
REG_TX_DST_IP_ADD_W0	0x20	0x20:reg_tx_dst_ip_add_w0 [31:0]
REG_TX_DST_IP_ADD_W1	0X24	0x24: reg_tx_dst_ip_add_w1 [63:32]
REG_TX_DST_IP_ADD_W2	0X28	0x28: reg_tx_dst_ip_add_w2 [95:64]
REG_TX_DST_IP_ADD_W3	0X2C	0x2C: reg_tx_dst_ip_add_w3 [127:96]
REG_IPV6_ADD_W0	0x30	0x30: reg_ipv6_add_w0[31:0]
REG_IPV6_ADD_W1	0X34	0x34: reg_ipv6_add_w1[63:32]
REG_IPV6_ADD_W2	0X38	0x38: reg_ipv6_add_w2[95:64]
REG_IPV6_ADD_W3	0X3C	0x3C: reg_ipv6_add_w3[127:96]
IPv6_SUBNET_PREFIX_LENGTH	0x40	0x40: [7:0] IPv6_SUBNET_PREFIX_LENGTH [31:8] reserved
REG_IPV6_GATEWAY_ADD_W0	0x44	0x44: reg_ipv6_gateway_add_w0[31:0]
REG_IPV6_GATEWAY_ADD_W1	0X48	0x48: reg_ipv6_gateway_add_w1[63:32]
REG_IPV6_GATEWAY_ADD_W2	0X4C	0x4C:reg_ipv6_gateway_add_w2[95:64]
REG_IPV6_GATEWAY_ADD_W3	0X50	0x50:reg_ipv6_gateway_add_w3[127:96]
REG_PAUSE_CONFIG	0x54	0x54: [15:0] pause_time. pause_cntrl. [31:17] reserved

Table 2.100. Enable IPv6 Register

Bit field	Name	Access	Width	description
0 31:1	reg_enable_ipv6	RW	1	0 means IPv4 is enabled by default. 1 means IPv6 is enabled by default.

Table 2.101. Destination MAC Address Register

Bit field	Name	Access	Width	description
31:0	reg_mac_add_l	RW	32	Destination MAC address is a unique 'hardware' address, which the user must choose for each instance. It is essential that this input matches the MAC address used by MAC interface.
15:0	reg_mac_add_h	RW	16	

Table 2.102. Destination IPv4 Address Register

Bit field	Name	Access	Width	description
31:0	reg_ipv4_add	RW	32	Destination IPV4 address is an address that is located in IP Header.

Table 2.103. Gateway IP Address Register

Bit field	Name	Access	Width	description
31:0	reg_gateway_add	RW	32	Gateway IP address is a gateway via which packets destined for a WAN are routed.

Table 2.104. Subnet Mask Register

Bit field	Name	Access	Width	description
31:0	reg_subnet_mask	RW	32	Subnet mask is used to determine whether an IP address is local (LAN) or remote (WAN).

Table 2.105. Destination Rx Port Number Register

Bit field	Name	Access	Width	description
15:0	reg_rx_port	RW	16	Destination Port number of UDP receiver.
31:16	RESERVED	RW	16	READ ALL 0

Table 2.106. Destination Tx Port Number Register

Bit field	Name	Access	Width	description
31:16	reg_tx_udp_dst_port	RW	16	Destination Port number of UDP transmitter.
15:0	reg_tx_udp_src_port	RW	16	Source Port number of UDP transmitter.

Table 2.107. Destination Tx IP Address Register

Bit field	Name	Access	Width	description
31:0	reg_tx_dst_ip_add_w0	RW	32	Destination IP address of UDP transmitter.
31:0	reg_tx_dst_ip_add_w0	RW	32	
31:0	reg_tx_dst_ip_add_w0	RW	32	
31:0	reg_tx_dst_ip_add_w0	RW	32	

Table 2.108. Destination IPv6 Address Register

Bit field	Name	Access	Width	description
31:0	reg_ipv6_add_w0	RW	32	Destination IPv6 address.
31:0	reg_ipv6_add_w1.	RW	32	
31:0	reg_ipv6_add_w2.	RW	32	
31:0	reg_ipv6_add_w3.	RW	32	

Table 2.109. Subnet Prefix Length Register

Bit field	Name	Access	Width	description
7:0	reg_ipv6_subnet_prefix_len	RW	8	Denotes how many bits of the address define the network in which it exists
31:8	Reserved	RW	24	

Table 2.110. IPv6 Gateway Address Register

Bit field	Name	Access	Width	description
31:0	reg_ipv6_gateway_add_w0	RW	32	Denotes how many bits of the address define the network in which it exists.
31:0	reg_ipv6_gateway_add_w1	RW	32	
31:0	reg_ipv6_gateway_add_w2	RW	32	
31:0	reg_ipv6_gateway_add_w3	RW	32	

Table 2.111. Pause Configuration Register

Bit field	Name	Access	Width	description
15:0	pause_time	RW	16	1-bit Pause control and 16 bits pause time
31:8	pause_cntrl	RW	1	

2.7. 2.7 Multiport Extension

Multiport IP with DDR and 4 FIFOs is a sub-system where multiple data streams can be handled simultaneously. It involves the use of DDR (Double Data Rate) memory for high-speed data storage and 4 FIFOs (First-In-First-Out) to manage the data flow. These FIFOs act as buffers, temporarily storing incoming data until it can be processed or transmitted. The DDR memory module is responsible for storing data in a high-speed, double data rate format. The overall architecture of Multiport Extension IP is shown in Figure 2.11.

The DDR memory is divided in two channels: One for Host and other for RISC -V CPU. The address and range for both channels can be configured using parameters: CH1_ADDR_RANGE, CH1_START_ADDR, CH2_ADDR_RANGE, and CH2_START_ADDR. CH2_START ADDR should start at least from CH1_ADDR_RANGE + CH1_START_ADDR.

Data in LPDDR4 is written in 64 Byte burst i.e., two transfers of 32 byte each in one transaction.

For more details on IP, refer to Multiport Extension IP user guide.

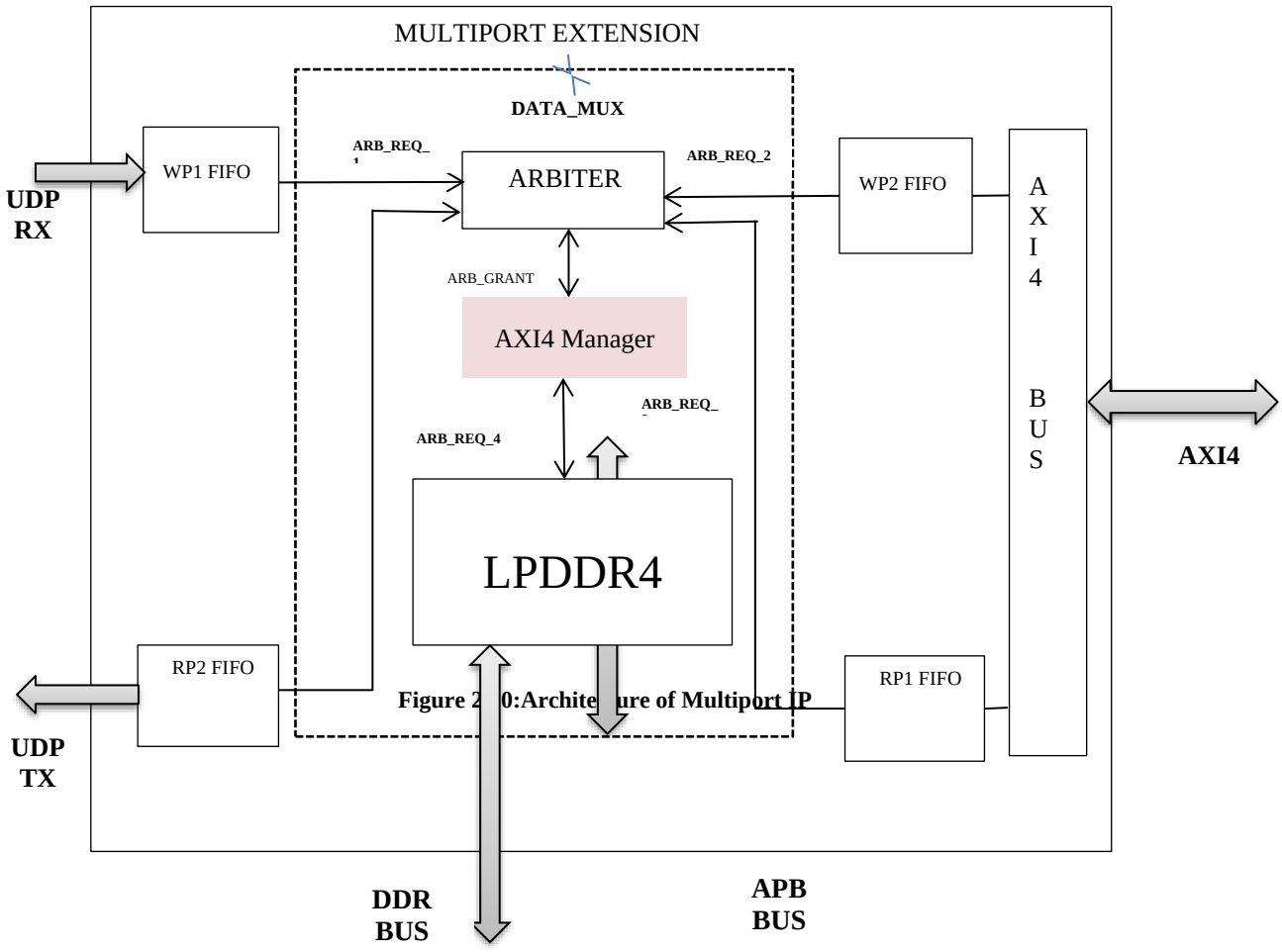

Figure 2.10:Architecture of Multiport IP

Figure 2.11. Register Map of Multiport extension IP

Table 2.112. Register Map of Multiport Extension

Register Name	Register Description	Offset	Access
WR_DATA_REG	Write Data Register	0x00	RW
CNTR_REG	Control Register	0x04	RW
RD_DATA_REG	Read Data Register	0x08	RO
STATUS	Status register	0x0C	RO
PCKT_COUNT_REG	Packet Count	0x10	RO

Table 2.113. Write Data Register

WR_DATA_REG				Base + 0x00
Byte	3	2	1	0
Name	WR_DATA_REG			
Default	0	0	0	0
Access	RW			

WR_DATA_REG[31:0]: RISC-V FIFO writes data

Table 2.114. Control Register

CNTR_REG				Base + 0x04
Byte	3	2	1	0
Name	CNTR_REG			
Default	0	0	0	0
Access	RW			

CNTR_REG[15:0]: Response Packet Length

CNTR_REG[16]: Write request

CNTR_REG[17]: Read request

CNTR_REG[31:18]: Reserved

Table 2.115. Read Data Register

RD_DATA_REG				Base + 0x08
Byte	3	2	1	0
Name	RD_DATA_REG			
Default	0	0	0	0
Access	RO			

RD_DATA_REG[31:0]: RISC-V FIFO write data

Table 2.116. Status register

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

STATUS[15:0]: Request Packet Length

STATUS[16]: UDP Write FIFO Full signal

STATUS[17]: UDP Write FIFO Empty signal

STATUS[18]: RISC-V Write FIFO Full signal

STATUS[19]: RISC-V Write FIFO Empty signal

STATUS[20]: RISC-V Read FIFO Full signal

STATUS[21]: RISC-V Read FIFO Empty signal

STATUS[22]: UDP Read FIFO Full signal

STATUS[23]: UDP Read FIFO Empty signal

STATUS[24] : Interrupt Info

STATUS[25]: Pll lock register

STATUS[31 : 26] : Reserved

Table 2.117. Packet Count

PCKT_COUNT_REG				Base + 0x10
Byte	3	2	1	0
Name	PCKT_COUNT_REG			
Default	0	0	0	0
Access	RO			

PCKT_COUNT_REG[31: 0] : UDP Packet Count

2.8. LPDDR4 Controller

Multi-port Extension IP is developed using existing LPDDR4 Memory Controller IP.

The LPDDR4 Memory Controller IP Core consists of three main blocks: Controller, Training engine, and PHY. There are two main interfaces: AXI4 and APB. AXI4 interface provides access to external memory to issue reads and writes. The APB interface provides interface to training engine and for error status-related registers for memory and configuration registers for training engine.

The Register Map of LPDDR4 is shown below:

LPDDR4 Initialization and Training:

- Enable initialization and training by writing to Training Operation Register. For simulation purposes, it is recommended to shorten the initialization by setting TRN_OP_REG.init_en = 0 and set 1 to all training enable bits. This step can be skipped when running on the actual FPGA device since the initialization and training are enabled by default.
- Write 2'h3 to Reset Register to un-reset the CPU and Memory Training Engine. Initialization and training begin after this step.
- Wait for completion of initialization and training by either of these steps:
- Wait for init_done_o signal assertion
- Poll Status Register until STATUS_REG.write_training done bit is asserted
- Wait for write_training done interrupt

The Steps for calibration are:

- Write the 0x20 register (training operation) as 0x1f, which is already by default 0x1f.
- After that read the 0x24 register (status).
- The register value at this point comes as 0x3001f, which means write_trn_done, read_trn_done, write_lvl_done, cbt_done, phy_ready all are asserted high. Also, the output port init_done_o is asserted high after this process.

The user can initiate AXI4 write and read access after the init_done_o signal asserts. Not all AXI4 accesses are supported. The proper translation of data from local data bus format to LPDDR4 format is not guaranteed when unsupported transaction is issued. The local databus access is converted to LPDDR4 BL16 or BL32 depending on the amount of data required by the local data bus burst access. The Memory Controller does not support unaligned addresses.

Table 2.118. Register Map of LPDDR4 Controller

Offset LMMI	Offset APB/AHBL	Register Name	Access Type	Description
0x0	0x00	WR_DATA_REG	WO	Write Data Register
0x0	0x00	RD_DATA_REG	RO	Read Data Register
0x1	0x04	SLV_SEL_REG	RW	Slave Select Register
0x2	0x08	CFG_REG	RW	Configuration Register
0x3	0x0C	CLK_PRESCL_REG	RW	Clock Pre-scaler Low Register
0x4	0x10	CLK_PRESCH_REG	RW	Clock Pre-scaler High Register
0x5	0x14	INT_STATUS_REG	RW1C	Interrupt Status Register
0x6	0x18	INT_ENABLE_REG	RW	Interrupt Enable Register
0x7	0x1C	INT_SET_REG	WO	Interrupt Set Register
0x8	0x20	WORD_CNT_REG	RO	Word Count Register
0x9	0x24	WORD_CNT_RST_REG	WO	Word Count Reset Register
0xA	0x28	TGT_WORD_CNT_REG	RW	Target Word Count Register
0xB	0x2C	FIFO_RST_REG	WO	FIFO Reset Register
0xC	0x30	SLV_SEL_POL_REG	RW	Slave Select Polarity Register
0xD	0x34	FIFO_STATUS_REG	RO	FIFO Status Register
0xE-0xF	0x38–0x3C	Reserved	RSVD	Reserved. Write access is ignored and 0 is returned on read access.

For more details, refer to Memory-Controller-IP-Core-Radiant-SW (FPGA-IPUG-02127).

2.9. SPI Flash Controller (QSPI Streamer)

This module is designed to stream data from external flash to FPGA using quad SPI data lines. It supports max 100 MHz for LFD2NX and LFCL devices. It has a prefetch buffer to enable cache feature for internal block of FPGA. Register space for this block is defined in register map section. This block does not have any configuration register to control it. There are basic settings (static configuration) which need to be selected during build generation. As per register map, complete SPI flash memory is directly accessible through AXI4 interface. AXI 4 interface supports both code and data reading features. This block does not support flash data write operation. This module is used in both main system and node system SOCs. This module only supports Micron and Macronix only.

In main system, it is used to steam instructions and static data into RISC-V from external SPI flash, and other parts of data section are stored in data ram.

At node end, it is used to stream instruction only into RISC-V from external SPI flash.

2.10. CNN Co-Processor Unit (CCU)

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

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

Table 2.119. CNN Co-Processor Unit Registers

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

Table 2.120. CNN Co-Processor unit control register

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

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

Table 2.121. CNN Co-Processor Unit Register

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

DONE:

1’b0: CCU process is NOT completed

1’b1: CCU process is completed

Table 2.122. Sign Select Configuration Register

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

SIGN_SEL: Sign selector of input and filter values

1'b0: Unsigned (TinyML HPD)

1'b1: Signed (ours)

Table 2.123. Input Offset Configuration Register

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

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

Table 2.124. Filter Offset Configuration Register

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

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

Table 2.125. Filter Offset Configuration Register

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

Table 2.126. Input Depth Configuration Register

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

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

Table 2.127. Input Data Address Configuration Register

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

INPUT_DATA_ADDR: Address to INPUT_DATA – start point of blob

Table 2.128. Filter Data Address Configuration Register

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

FILTER_DATA_ADDR: Address to FILTER_DATA – start point of filter

Table 2.129. CNN Co-Processor Unit Output Register

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

ACC_OUT: Accelerator output data

2.11. Motor Control and PDM Data Collector

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

This block is used only in the Node Systems. The top level of the Node System has an AHBL wrapper which has two AHBL subordinate ports. Mainly it consists of Motor Control and Predictive Maintenance (MC/PDM) Registers, Motor Control logic, and PDM Data Collector as shown in Figure 2.12.

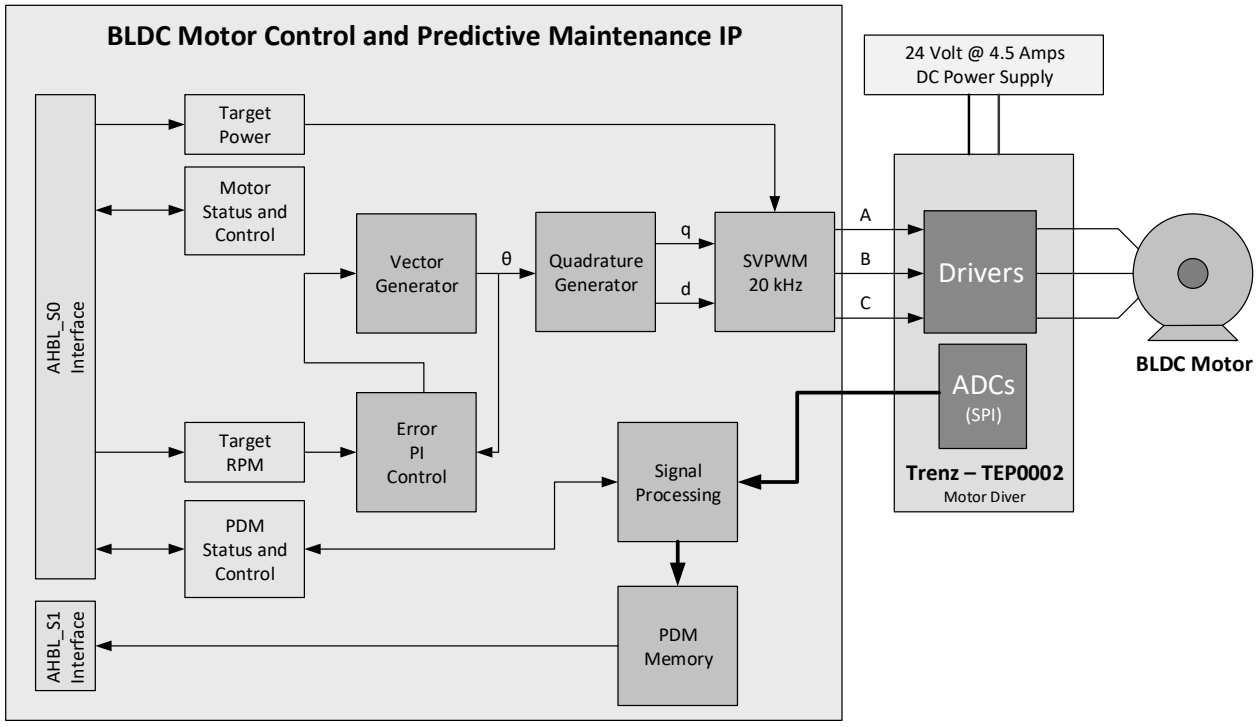

Figure 2.12. Motor Controller Interface with Motor

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

Table 2.130. Predictive Maintenance and Motor Control Registers

PDM/Motor Register Name	Register Function	Address	Access
MTRCR0	Motor Control Register 0 – Min RPM	Base + 0x00	Read/Write
MTRCR1	Motor Control Register 1 – Max RPM	Base + 0x04	Read/Write
MTRCR2	Motor Control Register 2 – RPM PI kI	Base + 0x08	Read/Write
MTRCR3	Motor Control Register 3 – RPM PI kP	Base + 0x0C	Read/Write
MTRCR4	Motor Control Register 4 – Torque PI kI	Base + 0x10	Read/Write
MTRCR5	Motor Control Register 5 – Torque PI kP	Base + 0x14	Read/Write
MTRCR6	Motor Control Register 6 – Sync Delay & Control	Base + 0x18	Read/Write
MTRCR7	Motor Control Register 7 – Target RPM	Base + 0x1C	Read/Write
MTRCR8	Motor Control Register 8 – Target Location	Base + 0x20	Read/Write
MTRCR9	Motor Control Register 9 – Location	Base + 0x24	Read/Write
MTRSR0	Motor Status Register 0 - RPM	Base + 0x28	Read
MTRSR1	Motor Status Register 1 – Limit SW & System Status	Base + 0x2C	Read
PDMCR0	Predictive Maintenance Control Register 0	Base + 0x30	Read/Write
PDMCR1	Predictive Maintenance Control Register 1	Base + 0x34	Read/Write
PDMSR	Predictive Maintenance Status Register	Base + 0x38	Read
PDMDDR	Predictive Maintenance ADC Data Register	Base + 0x3C	Read
PDMQDR	Predictive Maintenance ADC Data Register	Base + 0x40	Read
BRDSW	DIP and Push Button Switches	Base + 0x50	Read
BRDLEDS	LEDs and 7-Segment	Base + 0x54	Read/Write

Table 2.131. Motor Control 0 – Minimum RPM

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

MTRCR0[15:0]: MINRPM – Minimum RPM is the initial open loop motor starting RPM. Valid values are 10 to (216 -1).

MTRCR0[23:16]: MTRPOLES: Number of motor stator poles. Valid values are 1 to 255.

MTRCR0[31:24]: PI_DELAY: This is the RPM PI update rate. Valid values are 1 to 255.

Table 2.132. Motor Control 1 – Maximum RPM

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

MTRCR1[15:0]: MAXRPM – Maximum RPM is the upper limit RPM. Valid values are MINRPM to (216 -1).

MTRCR1[31:16]: TBD

Table 2.133. 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 – This is the gain of the Integrator part of the RPM PI control loop. Valid values are 1 to (216 -1).

MTRCR2[31:16]: RPMINT_MIN – Is the Integrator Anti-Windup Threshold. Valid values are 1 to (216 -1).

Table 2.134. Motor Control 3 – RPM PI Control Loop Proportional Gain (kP)

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

MTRCR3[15:0]: RPMPRPK – Is the gain of the Proportional part of the RPM PI control loop. Valid values are 1 to (216 -1).

MTRCR3[31:16]: RPMINT_LIM – Is the Integrator Anti-Windup Clamp. Valid values are 1 to (216 -1).

Table 2.135. Motor Control 4 – Torque PI Control Loop Integrator Gain (kI)

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

MTRCR4[15:0]: TRQINTK – Is the gain of the Integrator part of the Torque PI control loop. Valid values are 1 to (216 -1).

MTRCR4[31:16]: TRQINT_MIN – Is the Integrator Anti-Windup Threshold. Valid values are 1 to (216 -1).

Table 2.136. Motor Control 5 – Torque PI Control Loop Proportional Gain (kP)

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

MTRCR5[15:0]: TRQPRPK – Motor Power or Torque PI Proportional Gain, depends on value of MTRCR6[2].

MTRCR6[2] = 0: Motor Power - valid values are 0 to 1023.

MTRCR6[2] = 1: Torque PI Proportional Gain - valid values are 1 to (216-1).1

MTRCR5[31:16]: TRQINT_LIM – Is the Integrator Anti-Windup Clamp. Valid values are 1 to (216 -1).

Table 2.137. 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]: SYNCDLY1 – 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 (222 -1).

MTRCR6[23:22]: MTRCTRL_SYNDLYSF1 – Sync Delay Scale Factor

00 = Disable Sync Delay (single motor control or sync not used).

01 = Sync Delay Units is nano-seconds (10-9)

10 = Reserved

11 = Reserved

MTRCR6[24]: RESET_PI – Reset the RPM PI Control

0 = Normal Operation

1 = Force the output to match the input (zero input values force the output to default of 120 rpm)

MTRCR6[25]: STOP – Hold the Motor in Position

0 = Normal Operation

1 = Stop the motor rotation

MTRCR6[26]: TRQPI_MODE – Torque Control Mode controls how MTRCR5[15:0]: TRQPRPK is used:

0 = Open Loop Mode – TRQPRPK value specifies Motor Power.

1 = Closed Loop Mode – TRQPRPK value specifies the gain of the Proportional part of the Torque PI control loop.1

MTRCR6[27]: ESTOP – Emergency Stop

0 = Normal Operation.

1 = Engage E-Brakes without sync delay or MTR_ENGAGE.1

MTRCR6[28]: ENABLE – Enable Motor Drivers

0 = Disable Motor Drivers

1 = Enable Motor Drivers

MTRCR6[29]: MTR_MODE

0 = RPM Control – Slew to target RPM and continue to run until stop or change in RPM target

1 = Location Control – Rotate specified number of degrees or turns then stop. Ramp up from zero to Max RPM, run as needed, then ramp back down to zero.1

MTRCR6[30]: DIRECTION

0 = Clockwise Rotation

1 = Counter-Clockwise Rotation

MTRCR6[31]: ENGAGE – Sync Signal to latch all Control Registers from AHBL clock domain (50–100 MHz) to Motor clock domain (24–25 MHz). Write to all other control registers first (including this one with this bit off). Write to this register (read-modify-write) to set this bit. It can also be used to synchronize multiple nodes.

0 = No Updates to Motor or PDM Control registers.

1 = Transfer all control registers from AHBL holding registers to Motor PDM active registers.

Table 2.138. Motor Control Register 7 – Target RPM

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

MTRCR7[15:0]: TRGRPM – Target RPM. Valid values are 0 to (216 -1).

MTRCR7 [31:16]: tbd

Table 2.139. Motor Control Register 8 – Target Location

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

MTRCR8[31:0]: TRGLOC – Target Location. Valid values are -2,147,483,648 (-232) to 2,147,483,647 (232 -1).1

Approximately 24.8 hours @ 4,000 RPM counting each degree.

Table 2.140. Motor Control Register 9 – Current Location

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

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

Table 2.141. Motor Status Register 0 – RPM

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

MTRSR0[15:0]: MTRSTRPM – Current Motor RPM. Valid values are 0 to (216 -1).1

MTRSR0[31:16]: tbd.

Table 2.142. Motor Status Register 1

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

MTRSR1[0]: MTRSTR_MOV – Motor Moving
- 0 = Motor Stopped or coasting
- 1 = Motor Moving under control

MTRSR1[1]: ACCEL – Motor Accelerating
- 0 = Motor Not Accelerating
- 1 = Motor Accelerating

MTRSR1[2]: DECL - Motor Deaccelerating
- 0 = Motor Not Deaccelerating
- 1 = Motor Deaccelerating

MTRSR1[3]: RPM_LOCK - Motor at Target RPM
- 0 = Motor Not @ Target RPM
- 1 = Motor @ Target RPM

MTRSR1[4]: MTRSTR_STOP
- 0 = Motor not stopped
- 1 = Motor at zero RPM

MTRSR1[5]: MTRSTR_VLD_RPM
- 0 = RPM to Theta period calculation is still in process or invalid RPM request
- 1 = RPM to Theta period calculation is complete

MTRSR1[31:6]: tbd

Table 2.143. Predictive Maintenance Control Register 0

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

PDMCR0[0]: START – Start PDM data collection.
0 = Collection not started
1 = Collection started
PDMCR0[1]: PKDTEN – PDM Normalization Peak Detect Enable
0 = PDM Peak Detect is Disabled
1 = PDM Peak Detect is Enabled
PDMCR0[2]: FOLDEN – Enable Single Folding of PDM data
0 = Single Fold disabled
1 = Single Fold enabled
PDMCR0[3]: 2FOLDEN – Enable Double Folding of PDM data. All PDM training data was captured using Double Folding.
0 = Double Folding disabled
1 = Double Folding enabled
PDMCR0[4]: CONTINUOUS – Collect data as long as START = 1.
0 = Fixed – Collect PDM data for set number of rotations
1 = Continuous – Collect PDM data continuously (counting rotations in status reg)
PDMCR0[5]: TBD
PDMCR0[6]: CALIB – ADC offset calibration
0 = Normal operation
1 = Calibrate ADC offsets (motor not running)
PDMCR0[7]: ADCH – ADC Channel Select for PDMDDR and PDMQDR registers
0 = ADC Channel = Amps
1 = ADC Channel = Volts
PDMCR0[15:8]: PREREVS – Pre-Data Collection Revolutions
Number of Theta (Field Vector) revolutions to ignore before Data Collection. All PDM training data was captured using a value of 15.
PDMCR0[31:16]: DCREVS – Data Collection Revolutions
Theta (Field Vector) revolutions to capture PDM data (armature revs scale based on number of motor stator poles.
The motor used for training has 4-poles – 16 Theta rotations equate to four motor shaft rotations). Valid values 1 to 65,536.
All PDM training data was captured using 200 rotations.

Table 2.144. 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 2.145. Predictive Maintenance Status Register

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

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

Table 2.146. 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 A1

PDMDDR[31:16]: ADC1 Voltage or Current reading Phase B1

Table 2.147. 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 C1

PDMQDR[31:16]: ADC3 Voltage or Current reading of DC supply1

Table 2.148. Versa Board Switch Status Register

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

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

Table 2.149. Versa Board LED & PMOD Control Register

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

LED[0]: LED D18 – 0 = On, 1 = Off
LED[1]: LED D19 – 0 = On, 1 = Off
LED[2]: LED D20 – 0 = On, 1 = Off
LED[3]: LED D21 – 0 = On, 1 = Off
LED[4]: LED D22 – 0 = On, 1 = Off
LED[5]: LED D23 – 0 = On, 1 = Off
LED[6]: LED D24 – 0 = On, 1 = Off
LED[7]: LED D25 – 0 = On, 1 = Off
7SEG[0]: D36 Segment a – 0 = On, 1 = Off
7SEG[1]: D36 Segment b – 0 = On, 1 = Off
7SEG[2]: D36 Segment c – 0 = On, 1 = Off
7SEG[3]: D36 Segment d – 0 = On, 1 = Off

7SEG[4]: D36 Segment e – 0 = On, 1 = Off

7SEG[5]: D36 Segment f – 0 = On, 1 = Off

7SEG[6]: D36 Segment g – 0 = On, 1 = Off

7SEG[7]: D36 Segment dp – 0 = On, 1 = Off

PMOD2[0]: J8 Pin 1 I/O

PMOD2[1]: J8 Pin 2 I/O

PMOD2[2]: J8 Pin 3 I/O

PMOD2[3]: J8 Pin 4 I/O

PMOD2[4]: J8 Pin 7 I/O

PMOD2[5]: J8 Pin 8 I/O

PMOD2[6]: J8 Pin 9 I/O

PMOD2[7]: J8 Pin 10 I/O

PMOD2DIR[0]: J8 Pin 1 Direction – 0 = Input, 1 = Output

PMOD2DIR[1]: J8 Pin 2 Direction – 0 = Input, 1 = Output

PMOD2DIR[2]: J8 Pin 3 Direction – 0 = Input, 1 = Output

PMOD2DIR[3]: J8 Pin 4 Direction – 0 = Input, 1 = Output

PMOD2DIR[4]: J8 Pin 7 Direction – 0 = Input, 1 = Output

PMOD2DIR[5]: J8 Pin 8 Direction – 0 = Input, 1 = Output

PMOD2DIR[6]: J8 Pin 9 Direction – 0 = Input, 1 = Output

PMOD2DIR[7]: J8 Pin 10 Direction – 0 = Input, 1 = Output

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

2.12. SPI Manager IP Design Details

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

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

This section only provides minimum details on the SPI Manager IP required for integration and control. For more details, refer to SPI Manager IP user guide.

2.12.1. Overview

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

The SPI Manager IP provides a bridge between LMMI/AHB-Lite/APB and standard external SPI bus interfaces (functional diagram is shown in Figure 2.13. On the external, off-chip side the SPI Manager Controller IP has a standard SPI bus interface. On the internal, on-chip side, the SPI Manager Controller IP has LMMI/AHB-Lite/APB subordinate interface depending on the Interface attribute settings.

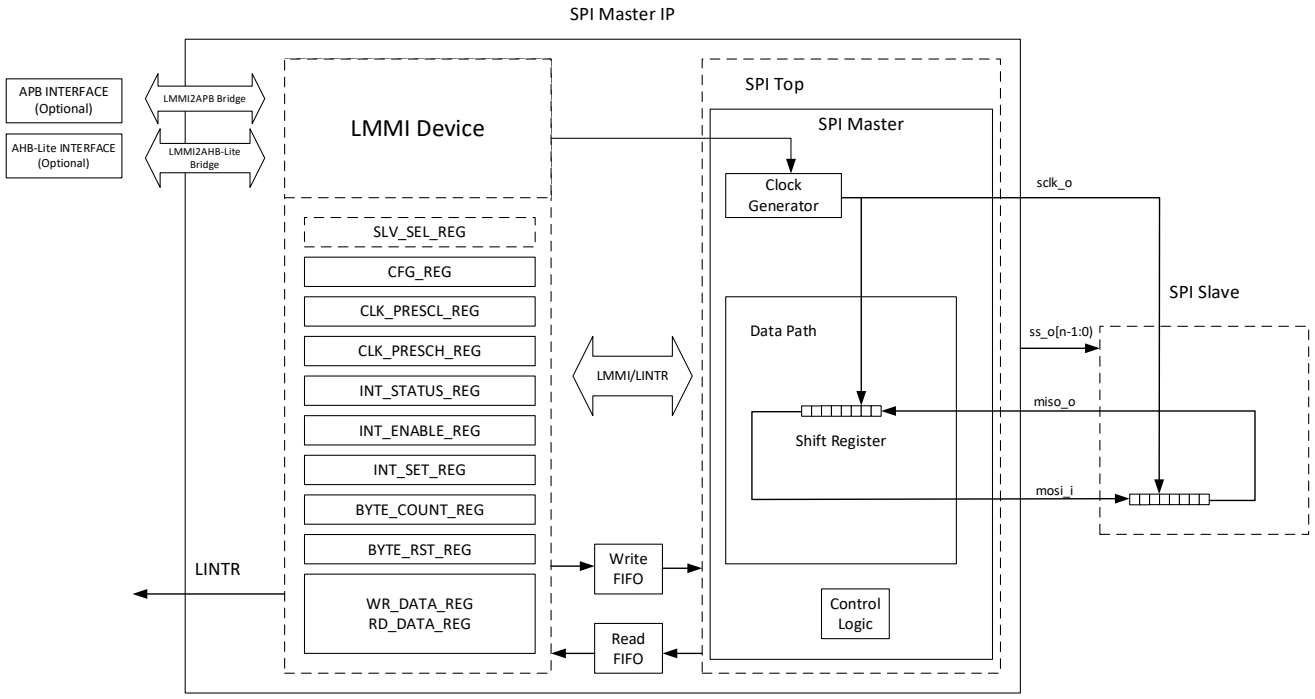

Figure 2.13. SPI Manager IP Core Block Diagram

2.12.2. SPI Manager Register Map

Table 2.150. SPI Manager Register Map

Offset LMMI	Offset APB/AHBL	Register Name	Access Type	Description
0x0	0x00	WR_DATA_REG	WO	Write Data Register
0x0	0x00	RD_DATA_REG	RO	Read Data Register
0x1	0x04	SLV_SEL_REG	RW	Subordinate Select Register
0x2	0x08	CFG_REG	RW	Configuration Register
0x3	0x0C	CLK_PRESCL_REG	RW	Clock Pre-Scaler Low Register
0x4	0x10	CLK_PRESCH_REG	RW	Clock Pre-Scaler High Register
0x5	0x14	INT_STATUS_REG	RW1C	Interrupt Status Register
0x6	0x18	INT_ENABLE_REG	RW	Interrupt Enable Register
0x7	0x1C	INT_SET_REG	WO	Interrupt Set Register
0x8	0x20	WORD_CNT_REG	RO	Word Count Register
0x9	0x24	WORD_CNT_RST_REG	WO	Word Count Reset Register
0xA	0x28	TGT_WORD_CNT_REG	RW	Target Word Count Register
0xB	0x2C	FIFO_RST_REG	WO	FIFO Reset Register
0xC	0x30	SLV_SEL_POL_REG	RW	Subordinate Select Polarity Register
0xD	0x34	FIFO_STATUS_REG	RO	FIFO Status Register
0xE 0xF	0x38-0x3C	Reserved	RSVD	Reserved. Write access is ignored and 0 is returned on read access.

Table 2.150 lists the address map and specifies the registers available to the user. The offset of each register is dependent on the Interface attribute setting as follows:

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

Note:

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

2.12.3. Programming Flow

2.12.3.1. Initialization

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

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

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

2.12.3.2. Transmit/Receive Operation

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

1. Set the following CFG_REG fields according to the target Subordinate settings: cpha, cpol, ssnp, and lsb_first. Set the only_write field based on the current transaction. If CFG_REG.only_write is 1'b0, SPI manager performs both transmit and receive operations (full-duplex). On the other hand, if CFG_REG.only_write is 1'b1, SPI Manager IP Core performs Transmit operation only.
2. Set TGT_WORD_CNT_REG according to the number of words to transfer.
3. Reset WORD_CNT_REG by writing 8'hFF to Word Count Reset Register
4. Write data words to WR_DATA_REG, amounting to ≤ FIFO Depth.
   Optional: If interrupt mode is desired, enable target interrupts in INT_ENABLE_REG If number of words to transfer is ≤ FIFO Depth, set tr_cmp_en = 1'b1. If number of words to transfer is > FIFO Depth, set the following: tx_fifo_aempty_en = 1'b1 and tr_cmp_en = 1'b1. Other interrupts not specified above are disabled.
5. If total number of words to transfer > FIFO Depth, wait for Transmit FIFO Almost Empty Interrupt.
   a. If polling mode is desired, read INT_STATUS_REG until tx_fifo_aempty_int asserts.
   b. If interrupt mode is desired, simply wait for interrupt signal to assert, then read INT_STATUS_REG and check that tx_fifo_aempt_int is asserted.
6. Clear Transmit FIFO Almost Empty Interrupt by writing 1'b1 to INT_STATUS_REG.tx_fifo_aempty_int. Clearing all interrupts by writing 8'hFF to INT_STATUS_REG is also okay since the user is not interested in other interrupts for this recommended sequence.
7. Write data words to WR_DATA_REG, amounting to less than or equal to (FIFO Depth – TX FIFO Almost Empty Flag).
8. If CFG_REG.only_write = 1'b0, read all the data in RD_DATA_REG. It is expected that Receive FIFO has (FIFO Depth – TX FIFO Almost Empty Flag - 1) amount of data words. Read INT_STATUS_REG.rx_fifo_ready_int to check if RD_DATA_REG is already empty.
9. If there is remaining data to transfer, go back to Step 6. Note that you can read Word Count Register to determine the number of words already transferred in SPI interface.
10. Wait for Transfer Complete Interrupt.
    a. If polling mode is desired, read INT_STATUS_REG until tr_cmp_int asserts.
    b. If interrupt mode is desired, set INT_ENABLE_REG = 8'h80 then wait for interrupt signal to assert. Then read INT_STATUS_REG and check that tr_cmp_int is asserted.
11. Clear all interrupts by writing 8'hFF to INT_STATUS_REG.
12. If CFG_REG.ONLY_WRITE = 1'b0, read all the data in RD_DATA_REG. Read INT_STATUS_REG.rx_fifo_ready_int to check if RD_DATA_REG is already empty

2.13. I²C Manager IP Design Details

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

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

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

2.13.1. Overview

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

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

Figure 2.14 shows the functional diagram of the I²C Manager Controller.

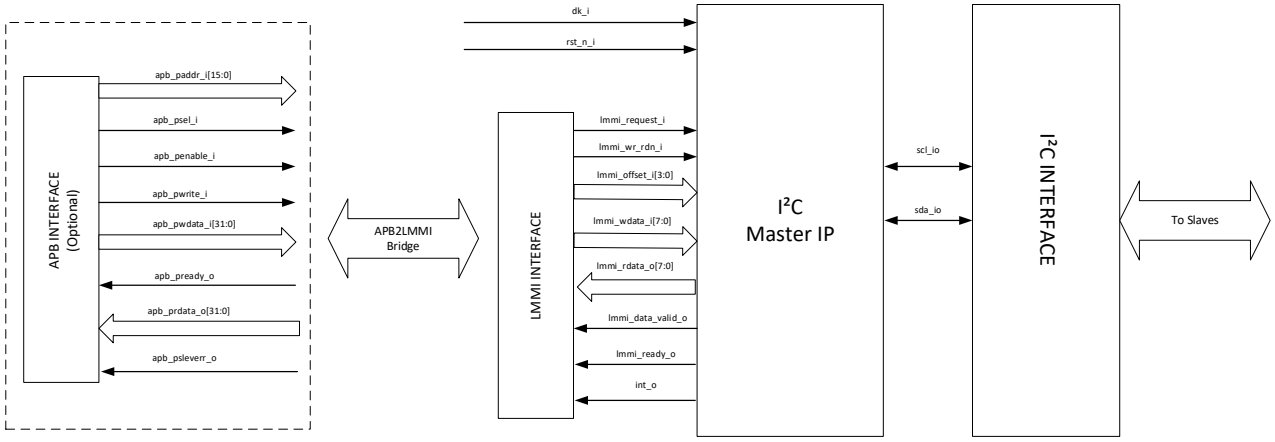

Figure 2.14. I²C Manager IP Core Functional Diagram

2.13.2. I²C Manager Register Map

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

Table 2.151 lists the address map and specifies the registers available to you. The offset of each register is dependent on attribute APB Mode Enable setting as follows:

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

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

Table 2.151. I²C Manager IP Core Registers Summary

Offset LMMI	Offset APB/AHBL	Register Name	Access Type	Description
0x0	0x00	WR_DATA_REG	WO	Write Data Register
0x0	0x00	RD_DATA_REG	RO	Read Data Register
0x1	0x04	SLAVE_ADDRL_REG	RW	Subordinate Address Lower Register
0x2	0x08	SLAVE_ADDRH_REG	RW	Subordinate Address Higher Register
0x3	0x0C	CONTROL_REG	WO	Control Register
0x4	0x10	TGT_BYTE_CNT_REG	RW	Byte Count Register
0x5	0x14	MODE_REG	RW	Mode Register
0x6	0x18	CLK_PRESCL_REG	RW	Clock Prescaler Low Register
0x7	0x1C	INT_STATUS1_REG	RW1C	First Interrupt Status Register
0x8	0x20	INT_ENABLE1_REG	RO	First Interrupt Enable Register

Offset LMMI	Offset APB/AHBL	Register Name	Access Type	Description
0x9	0x24	INT_SET1_REG	WO	First Interrupt Set Register
0xA	0x28	INT_STATUS2_REG	RW1C	Second Interrupt Status Register
0xB	0x2C	INT_ENABLE2_REG	RO	Second Interrupt Enable Register
0xC	0x30	INT_SET2_REG	WO	Second Interrupt Set Register
0xD	0x34	FIFO_STATUS_REG	RO	FIFO Status Register
0xE	0x38	SCL_TIMEOUT_REG	RW	SCL Timeout Register
0xF	0x3C	Reserved	RSVD	Reserved. Write access is ignored and 0 is returned on read access.

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

2.13.3. Programming Flow

2.13.3.1. Initialization

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

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

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

2.13.3.2. Writing to the Subordinate Device

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

To perform I2C write transaction:

1. Set the following MODE_REG fields according to the desired transfer mode: bus_speed_mode, addr_mode, ack_mode, clk_presc_high. Set the trx_mode field to 1'b0 for write transaction.
2. Set TGT_BYTE_CNT_REG according to the number of bytes to transfer.
3. Write data to WR_DATA_REG, amounting to ≤ FIFO Depth.
4. Set CONTROL_REG.start to 1'b1 to start the I²C transaction.
   Optional: If interrupt mode is desired, Enable target interrupts in INT_ENABLE1_REG. If number of words to transfer is ≤ FIFO Depth, set tr_cmp_en = 1'b1 If number of words to transfer is > FIFO Depth, set the following: tx_fifo_aempty_en = 1'b1 and tr_cmp_en = 1'b1. Other interrupts in this register are disabled.
5. If total number of bytes to transfer > FIFO Depth, wait for Transmit FIFO Almost Empty Interrupt. If polling mode is desired, read INT_STATUS1_REG until tx_fifo_aempty_int asserts. If interrupt mode is desired, simply wait for interrupt signal to assert, then read INT_STATUS1_REG and check that tx_fifo_aempt_int is asserted. In both cases, read also INT_STATUS2_REG to ensure that the transfer is good. I²C Manager IP Core
6. Clear Transmit Buffer Almost Empty Interrupt by writing 1'b1 to INT_STATUS1_REG.tx_fifo_aempty_int. Clearing all interrupts in this register by writing 8'hFF to INT_STATUS1_REG is also okay since the user is not interested in other interrupts for this recommended sequence.
7. Write data to WR_DATA_REG, amounting to less than or equal to (FIFO Depth – TX FIFO Almost Empty Flag).
8. If there is remaining data to transfer, go back to Step 6.
9. Wait for Transfer Complete Interrupt.

10. If polling mode is desired, read INT_STATUS1_REG until tr_cmp_int asserts. If interrupt mode is desired, set INT_ENABLE1_REG = 8'h80 then wait for interrupt signal to assert. Read INT_STATUS1_REG and if tr_cmp_int is asserted.
11. Clear all interrupts by writing 8'hFF to INT_STATUS1_REG.

2.13.3.3. Reading from the Subordinate Device

The following are the recommended steps for performing I^2C read transactions, assuming that the module is currently not performing any operation and if initialization is completed.

To perform I^2C read transaction:

1. Set the following MODE_REG fields according to the desired transfer mode: bus_speed_mode, addr_mode, ack_mode, clk_presc_high. Set the trx_mode field to 1'b1 for read transaction.
2. Set TGT_BYTE_CNT_REG according to the number of bytes to transfer.
3. Set CONTROL_REG.start to 1'b1 to start the I^2C transaction.
   Optional: If interrupt mode is desired, enable target interrupts in INT_ENABLE1_REG If number of words to transfer is ≤ FIFO Depth, set tr_cmp_en = 1'b1.
4. If number of words to transfer is > FIFO Depth, set the following: rx_fifo_afull_en = 1'b1 and tr_cmp_en = 1'b1. Other interrupts in this register are disabled.
5. If total number of bytes to receive > FIFO Depth, wait for Receive FIFO Almost Full Interrupt. If polling mode is desired, read INT_STATUS1_REG until rx_fifo_afull_int asserts. If interrupt mode is desired, wait for the interrupt signal to assert, and then read INT_STATUS1_REG and check if rx_fifo_afull_int is asserted. In both cases, read also INT_STATUS2_REG to ensure that the transfer is good.
6. Clear Receive FIFO Almost Full Interrupt by writing 1'b1 to INT_STATUS1_REG.rx_fifo_afull_int. Clearing all interrupts in this register by writing 8'hFF to INT_STATUS1_REG is also okay since the user is not interested in other interrupts for this recommended sequence.
7. Read all data from RD_DATA_REG. It is expected the amount of received data is less than or equal to (FIFO Depth – TX FIFO Almost Empty Flag). Read FIFO_STATUS_REG to confirm if Receive FIFO is emptied.
8. If there is remaining data to receive, go back to Step 5.
9. Wait for Transfer Complete Interrupt. If polling mode is desired, read INT_STATUS1_REG until tr_cmp_int asserts If interrupt mode is desired, set INT_ENABLE1_REG = 8'h80 and wait for the interrupt signal to assert. Read INT_STATUS1_REG and check that tr_cmp_int is asserted.
10. Clear all interrupts by writing 8'hFF to INT_STATUS1_REG.
11. Read all the remaining data from RD_DATA_REG.

2.14. UART IP Design Details

The Lattice Semiconductor UART (Universal Asynchronous Receiver/Transmitter) IP Core is designed for use in serial communication, supporting the RS-232.

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

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

2.14.1. Overview

The UART IP Core performs two main functions:

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

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

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

The registers of UART IP Core are accessed by the CPU (FPGA internal components) through an AMBA APB interface. The functional block diagram of UART IP Core is shown in Figure 2.15. The dashed lines in the figure are optional components/signals, which means they may not be available in the IP when disabled in the attribute.

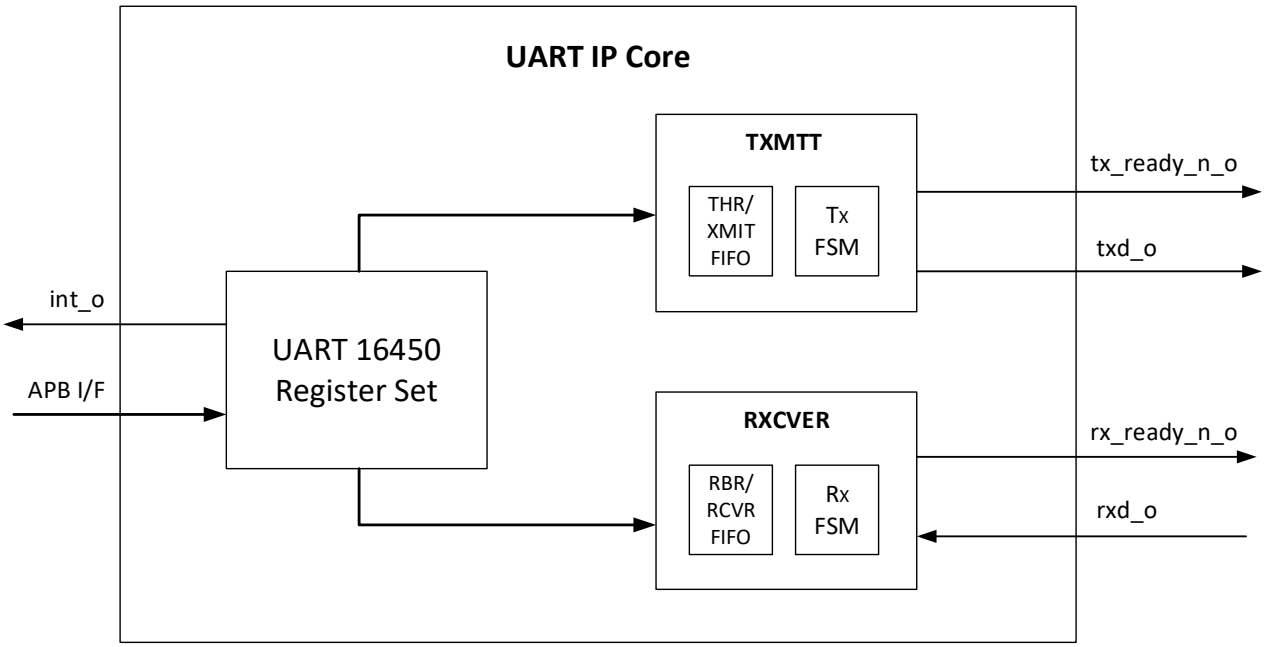

Figure 2.15. UART IP Core Functional Block Diagram

2.14.1.1. UART Register Description

The register address map, shown in Table 2.152, specifies the available IP Core registers. This is based on register set of UART 16450 but the offset address is changed to simplify the access to each register. The offset of each register increments by four to allow easy interfacing with the Processor and System Buses. In this case, each register is 32-bit wide wherein the lower 8 bits are used and the upper 24 bits are unused. The unused bits are treated as reserved – write access is ignored and read access returns 0.

Table 2.152. UART Register Map

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

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

2.14.2. Programming Flow

2.14.2.1. Initialization

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

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

These should match the corresponding setting in the communicating UART for the serial transaction to be successful.
Note that reset values of these register fields are configurable during IP generation. Thus, in some applications, initialization step is not necessary when attributes are properly set.

2.14.2.2. Transmit Operation

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

Transmit Operation – Interrupt Mode

To perform transmit operation in interrupt mode:

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

Transmit Operation – Polling Mode

To perform transmit operation in polling mode:

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

2.14.2.3. Receive Operation

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

Receive Operation – Interrupt Mode

To perform receive operation in interrupt mode:

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

Receive Operation – Polling Mode

To perform receive operation in polling mode:

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

2.14.2.4. Data Format

The character data written to THR and read from RBR is in little endian format as shown in Figure 2.16.

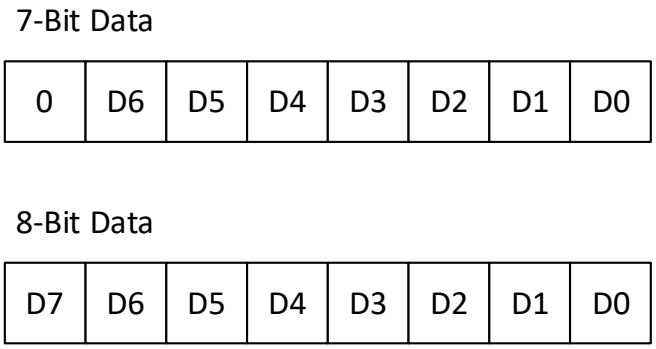

Figure 2.16. UART Data Format

3. Resource Utilization

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

Table 3.1. Main System Resource Utilization

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

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

Table 3.2. Node System Resource Utilization

Blocks	LUTs	EBRs	LRAMs	DSP MULT	Comments
RISC-V CPU	2537	2	—	—	—
ISR RAM	51	16	—	—	—
Data RAM (System Memory)	155	0	2	—	—
AHBL Interconnect	1721	—	—	—	—
APB Interconnect	14	—	—	—	—
FIFO DMA	754	16	—	—	—
EtherControl	4209	11	—	—	—
SPI Flash Controller	229	1	—	—	—
AHBL2APB	148	—	—	—	—
Motor Control Data Collector	4152	17	—	15.5	—
UART	261	—	—	—	—
I²C Manager	585	—	—	—	—
SPI Manager	398	—	—	—	—
Top-level	2	—	—	—	—
Total	**15216**	**63**	**2**	**15.5**	—

4. Software APIs

4.1. Main System APIs

4.1.1. Tasks of the Main System

The Main System acts as an interface between the user interface and the node system, which controls the motor IP. The commands are then sent to the nodes for configuration through EtherConnect. The Main System also enables the user interface to monitor various parameters of the motors. The system also receives commands from the GPIO switches attached to 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 EtherControl 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
  Users can Start, Stop, Accelerate, and Decelerate the motors with the help of switches provided. The Main System configures the node motor IP as per the switch configuration.
- Communicate with host system user interface over Ethernet
  The host system user interface sends configuration data and status check commands to the Main System, and the Main System responds based on the command.
- Communicate with Node System and motor IP over EtherConnect
  As per the commands received by the Main System, it creates burst packets to send to the Node System, which the Node System then receives and implements them. This communication between the main and Node System happens over EtherConnect and at a given time, a maximum of 256 bytes can only be transmitted from either direction.
- ISR3_EtherConnect
  *static void etherConnect_isr (void *ctx)*
- The primary function of the EtherConnect ISR function is to set the interrupt flag, acknowledge the interrupt, and return a value. The EtherConnect interrupt is used as an acknowledgment 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 in order to affirm the connection establishment with the Node System.
- Motor Configuration API
- *int motor_config_api(uint32_t address, uint32_t data, uint32_t multi)*
  This API is present in the main.c file. It needs three parameters namely:
  - address: signifies a register in the Motor Control IP
  - data: what needs to be written in that register
  - multi: data to be transmitted on multiple chains or selected chain only

It returns the following integer values:
- 0: if everything is correct
- –1: if there was any error

The API is called when there is a requirement to configure a register in the Motor Control IP of the Node System. This occurs in two cases:
- when there is an ON switch on any GPIO

- The API creates burst packets that 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: etherconnet packet to be transmitted on multiple chains or selected chain 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 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 has taken appropriate actions successfully, the Main System receives an interrupt and it validates the value of the interrupt info register. Upon the confirmation of the value of the interrupt info register, this API returns a 0 value or a –1 if there is an error.
- PDM Data Fetch API
- *int pdm_data_fetch_api(uint32_t total_size, uint32_t node_addr)*
- The API is present in the main.c file. It needs one parameter:
  - total_size: the size of the PDM data required from user interface
  - node_addr: node select value sent in packet
- It returns the following integer values:
  - 0: if all tasks are successfully completed
  - –1: if there is an error
- The API is called when there is a requirement to read a bulk maintenance data from the Motor Control IP of the Node System.
- The maximum data that can be transferred in a single transaction from node to Main System is 256 bytes. Therefore, if the total_size is larger than 256 bytes, chunks of 256 bytes are requested one by one until the total_size requirement is met.
- This API first configures the DMA register by writing the destination base and destination end address in specific registers. The API creates burst packets that 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 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 EtherConnect interrupt and it validates the value of the interrupt info register. The value of the DMA status register is to be validated as confirmation of the same. A successful validation signifies that a single chunk of data is successfully written into the Main System memory. This process is repeated until all the chunks are received by the Main System.
- A final EtherConnect interrupt is then received from the Node System signifying the completion of the PDM data fetches 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 the user interface
  - node_addr: node select value sent in packet
- It returns the following integer values:
  - 0: if all tasks are successfully completed
  - –1: if there is an error
- The API is called when there is a requirement to read a bulk maintenance data from the Motor Control IP of the Node System.
- This API is extended version of PDM Data Fetch API, as total size of data fetch depends on number of active nodes present in that chain.

4.1.2. OPCUA PubSub :

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

- OPC UA defines two different Network types for PubSub:
  - Local Network - which can use UDP Broadcast (or Unicast in some cases) or Ethernet APL. The messages are optimized binary UADP, which is defined in the OPC UA specifications. So, only OPC UA Subscribers can interpret the messages.
  - Message Queue Broker - 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.

Main System module implements following functions:

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

4.1.3. Create_UADP_NetworkMessage:

4.1.3.1. NetworkMessage Header:

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

Parameter of Network Message Header:

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

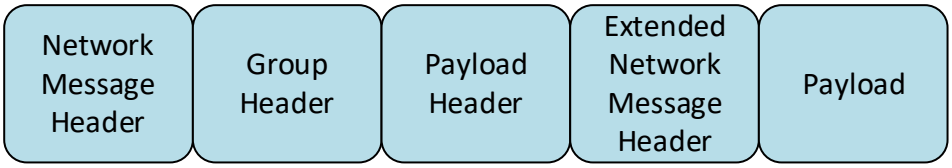

Figure 4.1: UADP Version

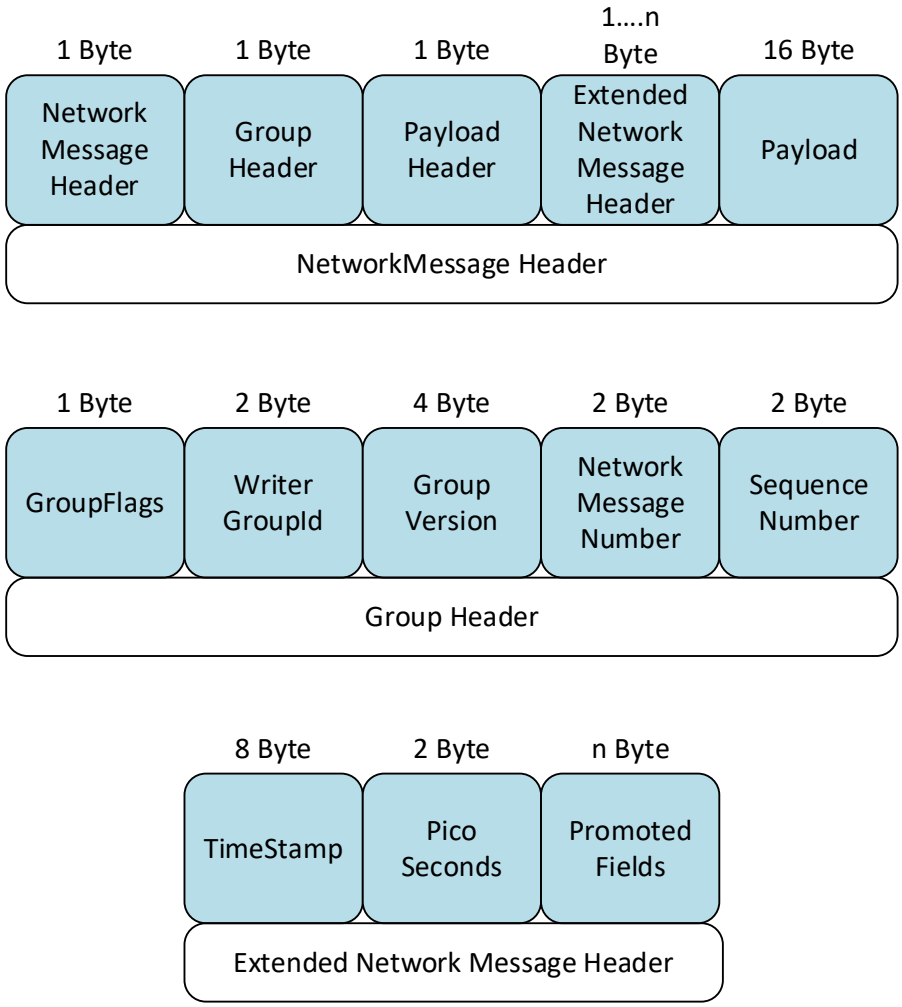

Figure 4.2: UADP Message packet header

4.1.4. GroupHeader:

The group header shall 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.

4.1.5. Extended NetworkMessage Header:

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

4.1.5.1. Payload

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

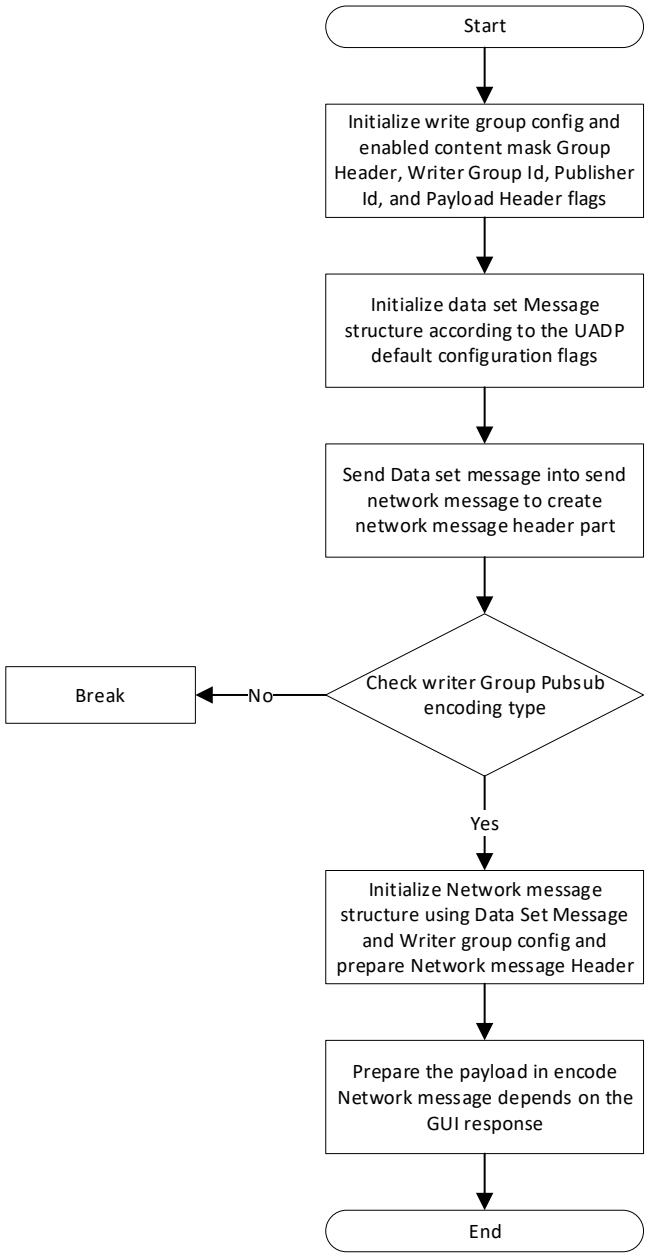

Figure 4.3: Create_UADP_NetworkMessage

UADP_NetworkMessage_parse:

This module parses the data received from the publisher. publisher sends the 64 Bytes opcua pubsub message, which holds the 20 bytes NetworkMessage header and, 44 Bytes payload. In payload data get the node ids and these node Ids are identify the method call or node variables or method variables, After identification creates an udp data reponse header,csv nodeid, request Type and value and writes 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 like startmotor , stop motor, power off, etc.

void UADP_NetWork_Parse(unsigned int *Buffer);

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

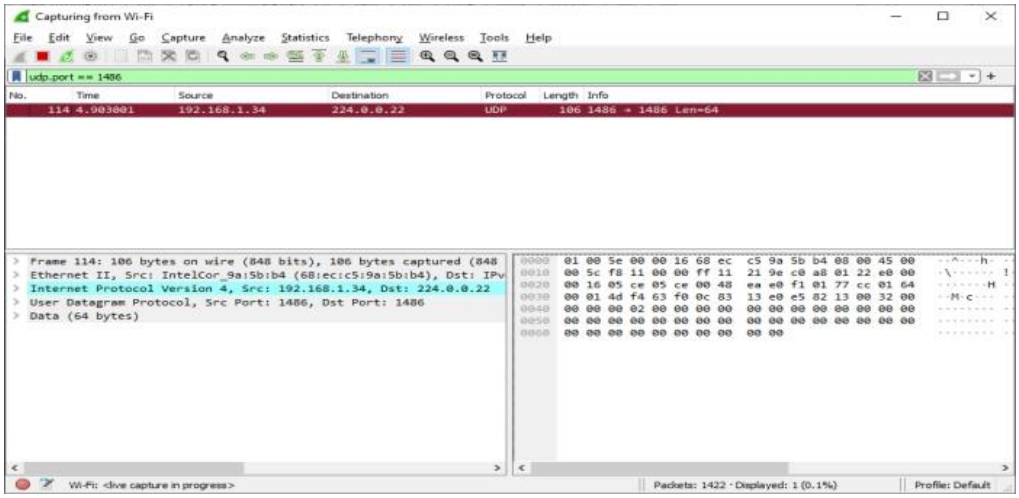

Figure 4.4: UADP Network message format

udp_response_func:

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

void udp_response_func ()

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

method_callbacks:

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

void method_callbacks(unsigned char method)

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

rfl_Update_config:

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

void rfl_Update_config()

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

Start_motor:

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

void Start_motor()

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

Stop_motor:

This function will stop motor of all nodes. This function will work when one of the motors is running.

void Stop_motor()

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

poweroff_motor:

This function will stop the power supply of all nodes. This function will work when one of the motors is running. This function will be disabled if all motors are off.

void poweroff_motor()

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

get_background:

This function will check Rpmlock, motor voltage and motor current status in background.

void get_background()

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

run_Pdm:

This moudle is collect the Pdm data to generate the the Pdm image.

void run_Pdm();

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

4.2. Node System APIs

4.2.1. Tasks of the Node System

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

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

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

4.2.2. Key Functions

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

- The primary function of the EtherConnect ISR function is to set the interrupt flag, acknowledge/clear the interrupt and return an integer value. The EtherConnect interrupts are used as indicators of the receipt of a command sent by the Main System to the Node System. The IRQ value for EtherConnect is 0.
- Node Configuration API
- *int node_config_api(void)*
- The API is present in the main.c file. It does not require any parameter.
- It returns the following integer values:
  - 0: if all tasks are successfully completed
  - –1: if there is an error
- The API is called when the main function receives a Node Config command in its Interrupt Status Register. This API reads the NODE ADDRESS register. This register contains an address of the peripheral (I2C, Modbus, SPI, and Motor IP) which is supposed to be configured. Next, the NODE CONFIG DATA register is read. This register has the configuration data. This data is then written into the address. If there is a read or write error, the API returns a –1 value. Once completed, the API returns a 0 value.
- Node Status API
- int node_status_api(void)
- The API is present in the main.c file. It does not require any parameter. This returns the following integer values:
  - 0: if all tasks are successfully completed
  - –1: if there is an error
- The API is called whenever the main function receives a Node Status command in its Interrupt Status Register. This API reads the NODE ADDRESS register. This register contains an address of the Node peripherial (Modbus, SPI, I2C, Motor IP) whose configuration value is supposed to be read. This address is then read and stored in a local variable data. This data is then written into the NODE STATUS register. If there is any read or write error, the API sends –1 value back. If everything goes okay, the API returns 0 value.
- PDM Data Fetch API
- int pdm_data_fetch_api(void)
- The API is present in the main.c file. It does not require any parameter. This returns the following integer values:
  - 0: if all tasks are successfully completed
  - –1: if there is an error
- The API first reads the size of PDM data required from the PDM ADDRESS register. It then writes the base address value and the end address (base address + size) value at the designated registers in the FIFO DMA Module. It then enables the FIFO DMA module by sending writing 0x00000003 first and then 0x00000000 to the FIFO DMA CONTROL register. Once done, it polls the DMA STATUS register for the indication of completion of the PDM data fetch. Once it receives the done value, it sets the DMA DONE INDICATE register. If there is any read or write error, the API sends –1 value back. If everything goes okay, the API returns 0 value.
- Node Peripheral APIs
  - I^2C Master

    The following are the I^2C BSP functions used in the main.c file for writing and reading the I^2C 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 the function 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 fetches 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)

5. Communications

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

5.1. Communication between Host and Main System

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

5.1.1. Messages from Host to Main System

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

5.1.2. Messages from Main System to Host

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

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

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

5.2.1. Messages from Main System to Node System

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

5.2.2. Messages from Node System to Main System

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

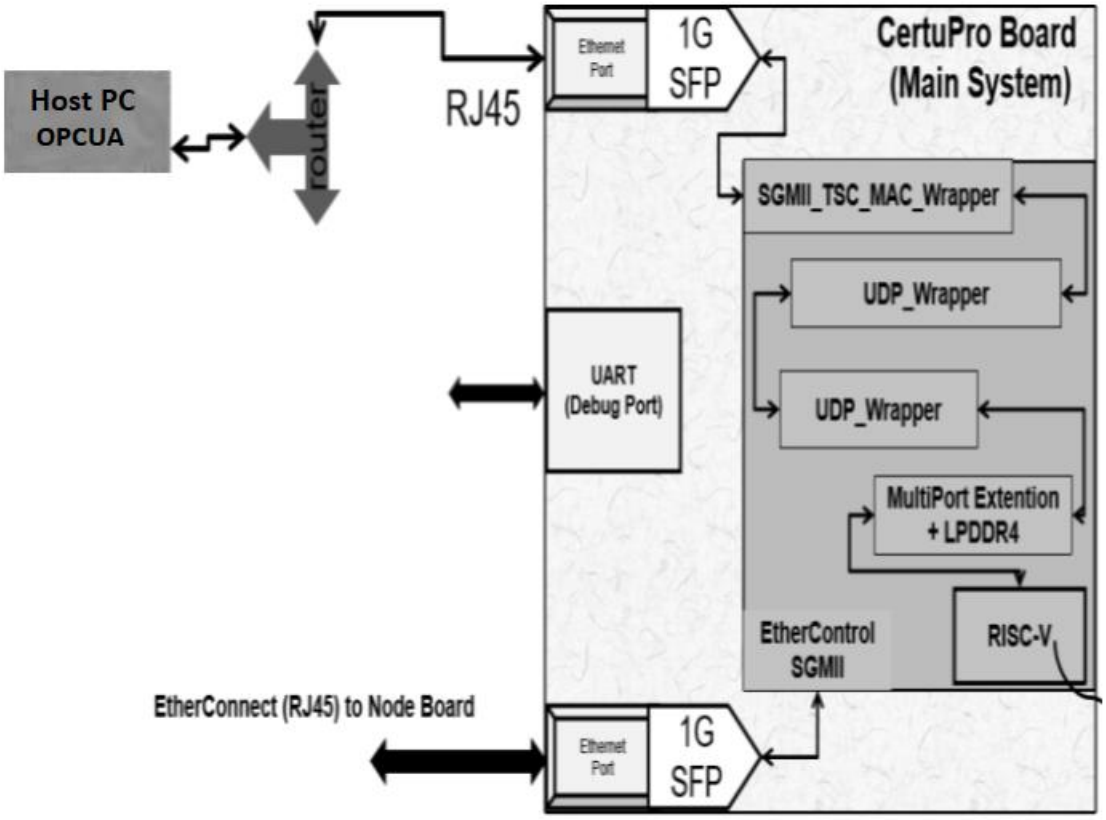

Figure 5.1: Data flow from Host to Node system via Main system

Appendix A: Predictive Maintenance with TensorFlow Lite

A.1 Introduction

The predictive maintenance is supported in Automate stack using trained neural network. Currently, the network is trained with data collected from good and bad BLDC motors. We further provide scripts and models for users to train the network with their own dataset. This section describes how models can be trained using TensorFlow framework on a GPU machine and later translated to a runtime object for deployment on main system RISC-V for inferencing using TensorFlow-Lite converter.

The ADC on the motor control board reads the current as data and then this data is transformed using Clark transform equation shown below. The transformed data when plotted will show as a circle. Any distortion in this circle gives the indication that motor is not operating normally. Note that the data collected in the GUI and used for training is a double folded version of this plot. This folding helps to reduce sparsity in the image and reduce amount of data that needs to be transferred from node to main system.

$$iD = \left(\frac{\sqrt{2}}{\sqrt{3}}\right) iA - \left(\frac{1}{\sqrt{6}}\right) iB - \left(\frac{1}{\sqrt{6}}\right) iC \; and \; iQ = \left(\frac{1}{\sqrt{2}}\right) iB - \left(\frac{1}{\sqrt{2}}\right) iC$$

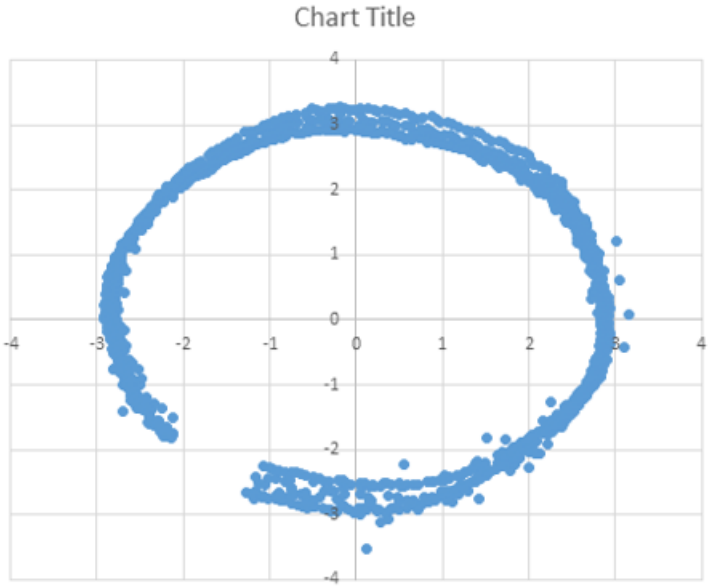

Figure A.1. Clark Equation and the plot

Below is bad image generated from PDM data of a broken motor:

Figure A.2. PDM Data Collected from a broken motor

Below is good image generated from PDM data of good motor:

Figure A.3. PDM Data Collected from broken motor

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

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

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

A.1.1.1. Installing the CUDA Toolkit

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

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

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

Figure A.4. Download CUDA Repo

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

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

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

Figure A.5. Install CUDA Repo

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

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

Figure A.6. Fetch Keys

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

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

Figure A.7. Update Ubuntu Packages Repositories

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

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

Figure A.8. CUDA Installation

A.1.1.2. Installing the cuDNN

To install the cuDNN:

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

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

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

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

Figure A.9. cuDNN Library Installation

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

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

A.1.2.1. Installing the Anaconda Python

To install the Anaconda and Python 3:

1. Go to https://www.anaconda.com/products/individual#download web page.
2. Download Python3 version of Anaconda for Linux.
3. Run the command below to install the Anaconda environment:

```
$  sh  Anaconda3-2019.03-Linux-x86_64.sh
```

Note: Anaconda3-**<version>**-Linux-x86_64.sh, version may vary based on the release.

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

Welcome to Anaconda3 2020.07

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

Figure A.10. Anaconda Installation

4. Accept the license.

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

Figure A.11. Accept License Terms

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

```
[no] >>> yes


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

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

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

Figure A.12. Confirm/Edit Installation Location

6. After installation, enter *no*.

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

Figure A.13. Launch/Initialize Anaconda Environment on Installation Completion

A.1.3. Installing the TensorFlow version 1.15

To install the TensorFlow version 1.15:

1. Activate the Anaconda environment by running the command below:

```
$ source <conda directory>/bin/activate
```

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

Figure A.14. Anaconda Environment Activation

2. Install the TensorFlow by running the command below:

```
$ conda install tensorflow-gpu==1.15.0
```

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

## Package Plan ##

  environment location: /home/user/anaconda3/

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

Figure A.15. TensorFlow Installation

3. After installation, enter Y.

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


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

Figure A.16. TensorFlow Installation Confirmation

TensorFlow installation is complete.

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

Figure A.17. TensorFlow Installation Completion

A.1.4. Installing the Python Package

To install the Python package:

1. Install Easydict by running the command below:

```
$ conda install -c conda-forge easydict
```

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

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - easydict
```

Figure A.18. Easydict Installation

2. Install Joblib by running the command below:

```
$ conda install joblib
```

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

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - joblib
```

Figure A.19. Joblib Installation

3. Install Keras by running the command below:

```
$ conda install keras
```

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

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - keras
```

Figure A.20. Keras Installation

4. Install OpenCV by running the command below:

```
$ conda install opencv
```

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

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - opencv
```

Figure A.21. OpenCV Installation

5. Install Pillow by running the command below:

```
$ conda install pillow
```

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

## Package Plan ##

  environment location: /home/user/anaconda3

  added / updated specs:
    - pillow
```

Figure A.22. Pillow Installation

A.2. Creating the TensorFlow Lite Conversion Environment

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

1. Create a new Anaconda environment.

```
$ conda create -n <New Environment Name> python=3.6
```

2. Activate the newly created environment.

```
$ conda activate <New Environment Name>
```

3. Install Tensorflow 2.2.0.

   Note: We have noticed output differences in Tensorflow(2.2.0) and Tensorflow-gpu(2.2.0) in terms of tflite size. It is recommended to use TensorFlow (2.2.0).

```
$ conda install tensorflow=2.2.0
```

4. Install opencv.

```
$conda install opencv
```

A.3. Preparing the Dataset

This section describes the steps and guidelines used to prepare the dataset for training the predictive maintenance.

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

A.3.1. Dataset Information

In the predictive maintenance demonstration, there are three classes: bad, Normal, and unknown. The dataset should be organized as shown in Figure A.23,contains bad motor data and 1 contains normal motor data.

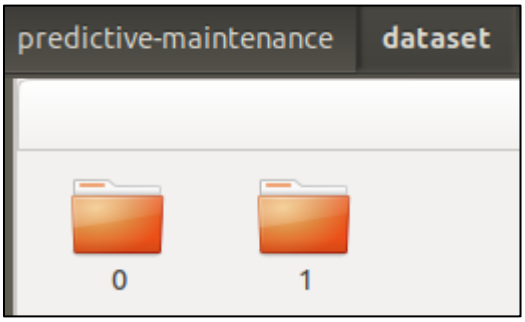

Figure A.23. Predictive Maintenance Dataset

A.4. Preparing the Training Code

Notes:

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

A.4.1. Training Code Structure

Download the Lattice predictive maintenance demo training code. Figure A.24 shows the directory structure.

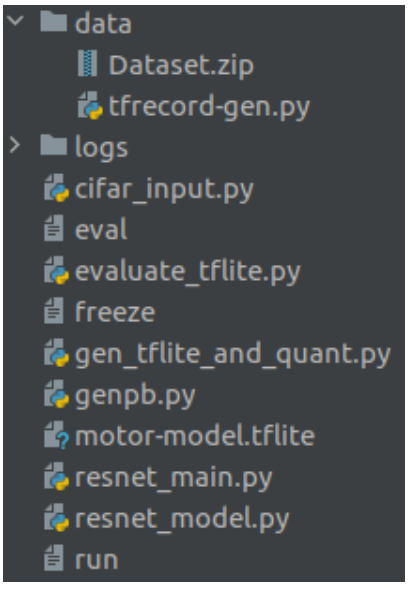

Figure A.24. Training Code Directory Structure

A.4.2. Generating tfrecords from Augmented Dataset

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

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

The input directory should follow the structure shown in Figure A.24.

A.4.3 Neural Network Architecture

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

Table A.1. Predictive Maintenance Training Network Topology

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

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

- Layer information
  - Convolutional Layer:
    In general, the first layer in a CNN is always a convolutional layer. Each layer consists of number of filters (sometimes referred as kernels) which convolve with input layer/image and generate an 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 of 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 2×2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every sub-region that the filter convolves around.

  The intuitive reasoning behind this layer is that once the user knows that a specific feature is in the original input volume (a high activation value results), its exact location is not as important as its relative location to the other features. As you can imagine, this layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes. The first is that the number of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it controls overfitting. 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 overfitting 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. In order to train a neural network, perform pre-processing to the input data. For example, the user can normalize all data so that it resembles a normal distribution (that means, zero mean and a unitary variance). Reason being preventing the early saturation of non-linear activation functions like the sigmoid function, assuring that all input data is in the same range of values, 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 in order 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 doesn't perform well when given new examples. The idea of dropout is simplistic in nature. This layer drops out a random set of activations in that layer by setting them to zero. It forces the network to be redundant. That means the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network is not getting too "fitted" to the training data and thus helps alleviate the over fitting problem. An important note is that this layer is only used during training, and not during test time.
- Fully connected Layer

  This layer basically takes an input volume (whatever the output is of the conv or ReLU or pool layer preceding it) and outputs an N dimensional vector where N is the number of classes that the program must choose from.
- Quantization

  Quantization is a method to bring the neural network to a reasonable size, while also achieving high performance accuracy. This is especially important for on-device applications, where the memory size and number of computations are necessarily limited. Quantization for deep learning is the process of approximating a neural

network that uses floating-point numbers by a neural network of low bit width numbers. This dramatically reduces both the memory requirement and computational cost of using neural networks.

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

A.4.4. Training Code Overview

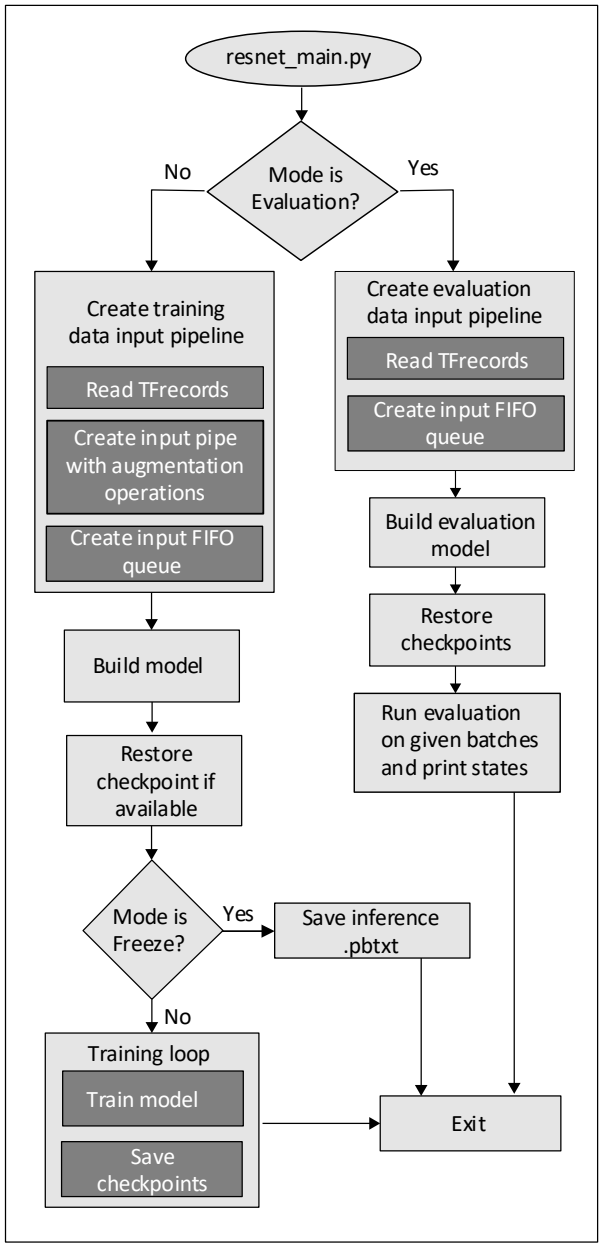

Figure A.25. Training Code Flow Diagram

A.4.4.1. Configuring Hyper-Parameters

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

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

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

Figure A.26. Code Snippet: Hyper Parameters

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

A.4.4.2. Creating Training Data Input Pipeline

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

Figure A.27. Code Snippet: Build Input

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

Read tfrecords

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

Figure A.28. Code Snippet: Parse tfrecords

- The above snippet reads tfrecord files and parses its features which are height, width, label, and image.

Converting Image to Grayscale and Scaling the Image

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

Figure A.29. Code Snippet: Convert Image to Gray Scale

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

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

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

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

Figure A.30. Code Snippet: Convert Image to Gray Scale

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

Creating Input Queue

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

Figure A.31. Code Snippet: Create Queue

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

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

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

Figure A.32. Code Snippet: Add Queue Runners

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

A.4.4.3. Model Building

CNN Architecture

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

Figure A.33. Code Snippet: Create Model

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

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

Figure A.34. Code Snippet: Fire Layer

Arguments of *_vgg_layer*:

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

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

        return pool
```

Figure A.35. Code Snippet: Convolution Block

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

Feature Depth of Fire Layer

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

Figure A.36. Code Snippet: Feature Depth Array for Fire Layers

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

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

Figure A.37. Code Snippet: Forward Graph Fire Layers

Loss Function and Optimizers

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

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

Figure A.38. Code Snippet: Loss Function

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

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

Figure A.39. Code Snippet: Optimizers

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

A.4.4.4. Restore Checkpoints

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

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

Figure A.40. Code Snippet: Restore Checkpoints

A.4.4.5. Saving .pbtxt

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

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

Figure A.41. Code Snippet: Save .pbtxt

A.4.4.6. Training Loop

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

Figure A.42. Code Snippet: Training Loop

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

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

Figure A.43. Code Snippet: LearningRateSetterHook

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

- Summary_hook

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

Figure A.44. Code Snippet: Save Summary for Tensorboard

- Saves tensorboard summary for every 100 steps.

- Logging_hook

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

Figure A.45. Code Snippet: logging hook

- Prints logs after every 100 iterations.

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

A.4.5.1. Training

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

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

Figure A.46. Predictive Maintenance – Run Script

To start training run the run script as mentioned below.

```
$ ./run
```

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

Figure A.47. Predictive Maintenance – Trigger Training

A.4.5.2. Transfer Learning

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

Figure A.48. Predictive Maintenance – Trigger Training with Transfer Learning

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

A.4.5.3. Training Status

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

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

Figure A.49. Predictive Maintenance – Training Logs

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

Figure A.50. Predictive Maintenance – Confusion Matrix

Use TensorBoard utility for checking training status.

- Start TensorBoard by below command:

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

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

Figure A.51. TensorBoard – Launch

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

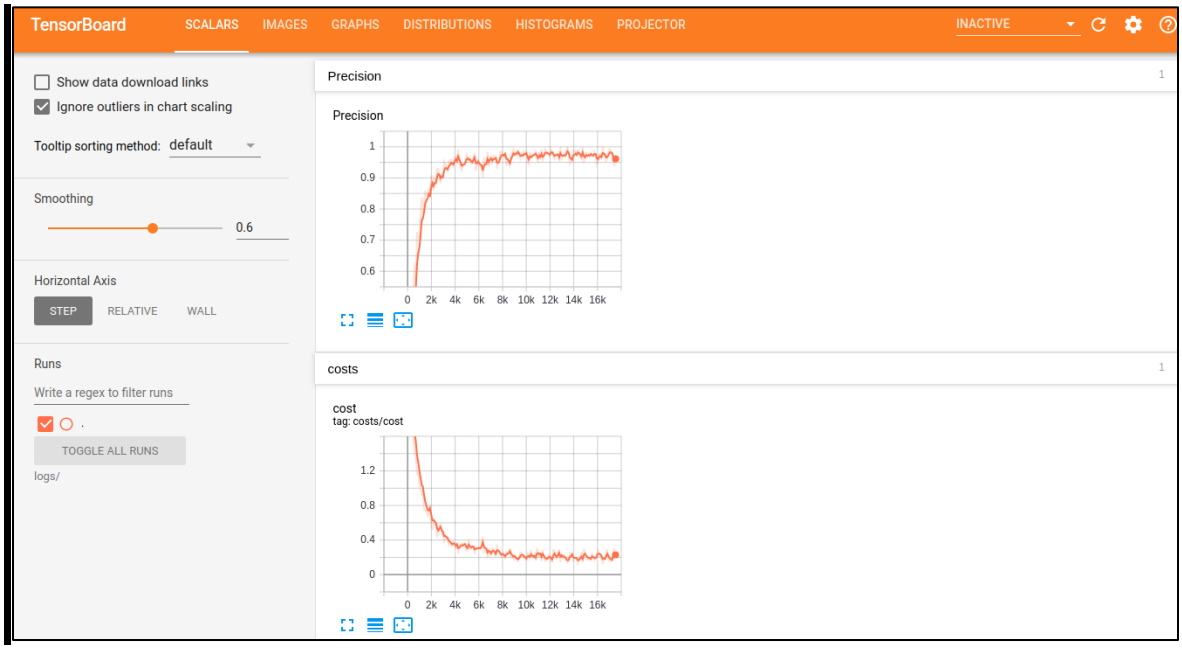

Figure A.52. TensorBoard – Link Default Output in Browser

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

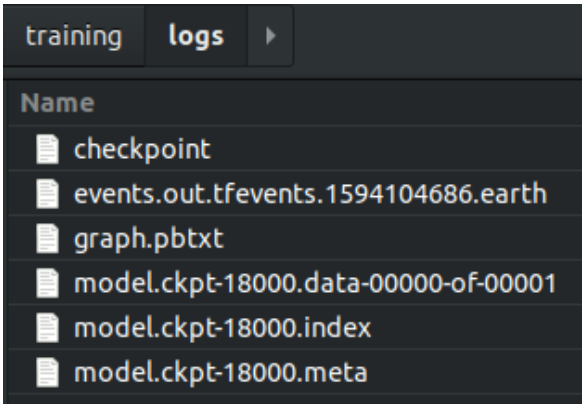

Figure A.53. Checkpoint Storage Directory Structure

A.5. Creating Frozen File

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

A.5.1. Generating .pbtxt File for Inference

Once the training is completed run below command to generate inference .pbtxt file.

Note: Do not modify config.sh after training.

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

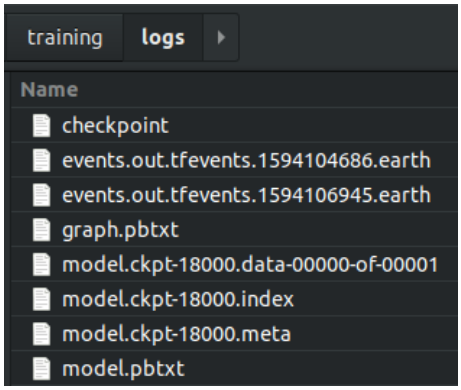

Figure A.54. Generated '.pbtxt' for Inference

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

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

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

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

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

Figure A.55. Run genpb.py To Generate Inference .pb

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

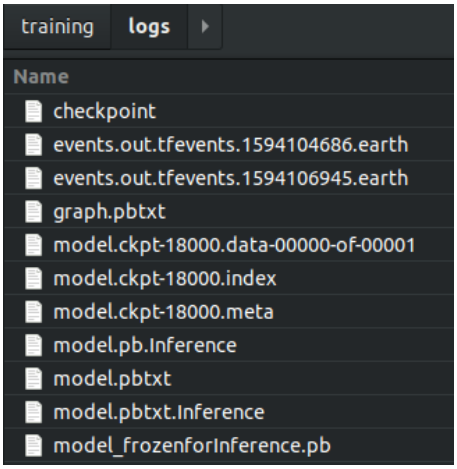

Figure A.56. Frozen Inference .pb Output

A.6. TensorFlow Lite Conversion and Evaluation

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

Note: It is recommended to use Tensorflow 2.2.0 (CPU Only) instead Tensorflow 1.15.0 In TensorFlow Lite conversion flow.

Use Environment created from the Creating the TensorFlow Lite Conversion Environment section.

A.6.1. Converting Frozen Model to TensorFlow Lite

User can find “gen_tflite_and_quant.py” under training code which converts frozen model to TensorFlow Lite and also quantize it with INT8 quantization.

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

Arguments information:

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

The command saves TensorFlow Lite at given path.

A.6.2. Evaluating TensorFlow Lite model

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

Argument information:

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

The command shows accuracy on both classes.

A.6.3. Converting TensorFlow Lite To C-Array

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

The command generates c array at path given by user.

For detailed instructions on setting compiling the code, installing the client-end application, automating stack 3.0 bit file and generating binary, programming the Automate Stack on SPI Flash memory, troubleshooting the main system board, debugging using Docklight, OPCUA Modeler, and CSV file, refer to Automate Stack 3.0 Demo User Guide (FPGA-UG-02164).

Appendix B: Setting up the Wireshark tool

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

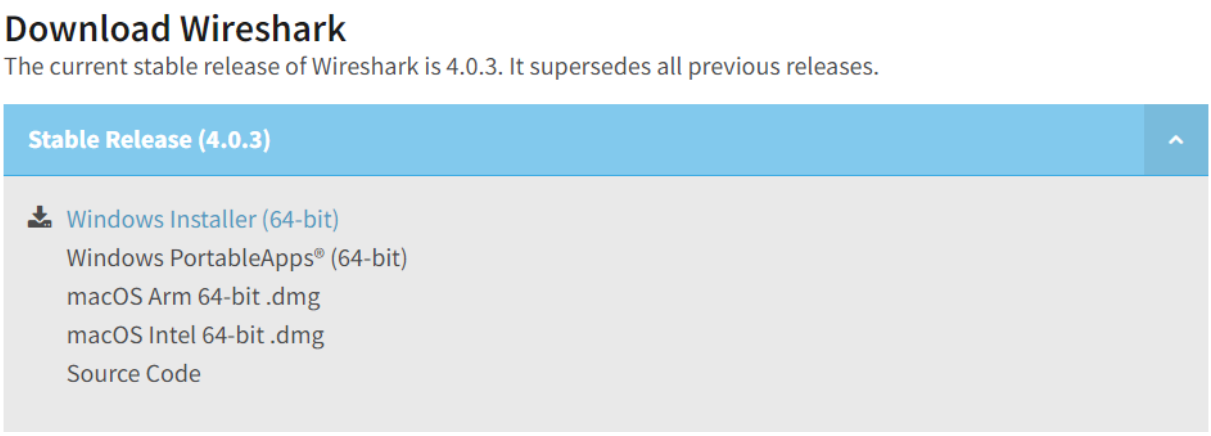

Figure B.1. Downloadable link of Wireshark

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

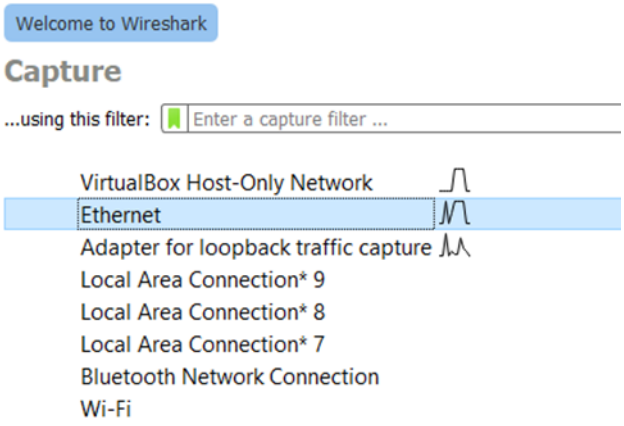

Figure B.2. Wireshark tool: Ethernet selection

3. Now click on the run( ) button.
4. Check udp message use port filter (udp.port == 1486) on the top bar

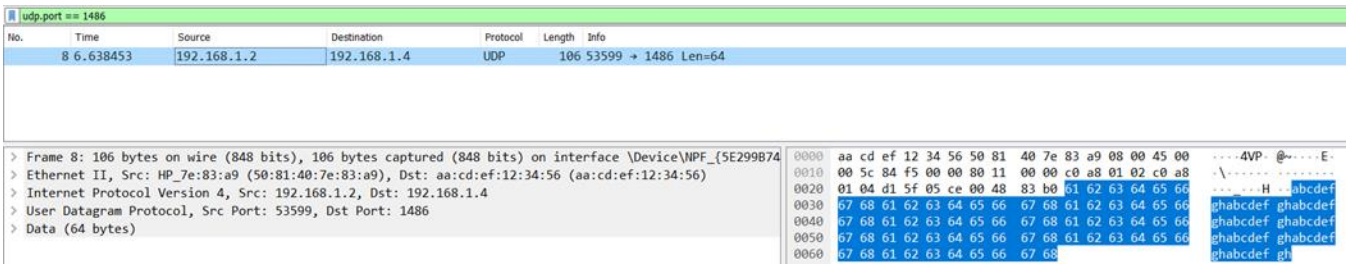

Figure B.3. Wireshark tool - write udp.port == 1486

5. Check the both Source and destination IP.

udp

No.	Time	Source	Destination	Protocol	Length	Info
2690…	6.993534	192.168.1.2	192.168.1.4	UDP	106	53599 → 1486 Len=64
2691…	7.305692	192.168.1.4	192.168.1.2	UDP	106	1500 → 1482 Len=64

Figure B.4. Source and Destination udp packet

6. Click on the UDP packet

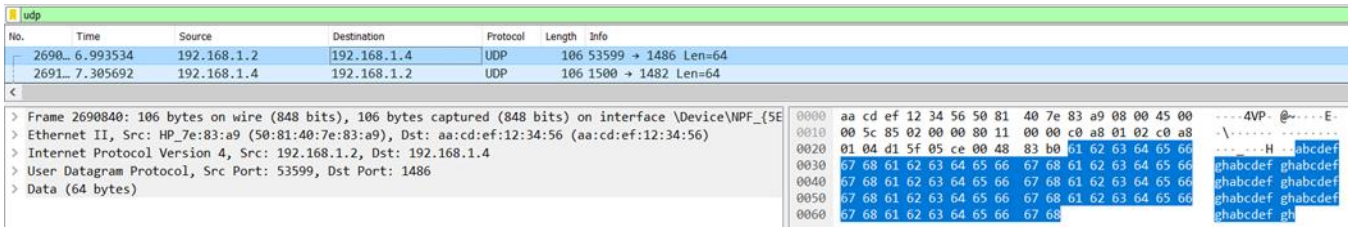

Figure B.5. Wireshark tool - first udp packet

Appendix C: Automate Stack 3.0 Bitfile and Binary Generation

C.1 Steps for Bit File Generation

C.1.1 MAIN SYSTEM

1. Open Generated Radiant Project in Radiant Tool.
2. Select Family: LFCPNX
3. Select Device: LFCPNX-100
4. Select Operating Condition: Industrial
5. Select Package: LFG672
6. Performance Grade: 9_High-Performance_1.0V
7. Part Number: LFCPNX-100-9LFG672I

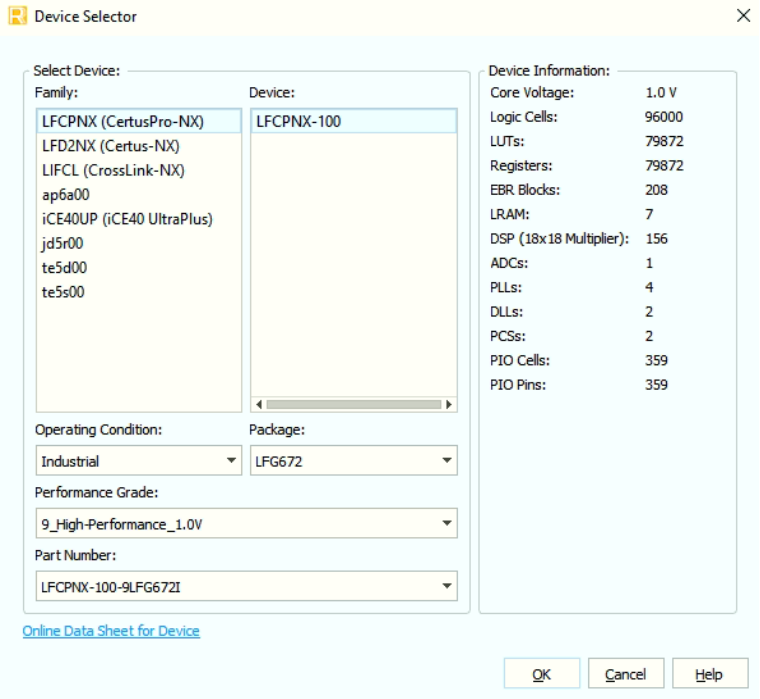

Figure C.1. Lattice Radiant Device Selector for Main System

8. Change strategy as shown below: Frequency Parameter 250 MHz

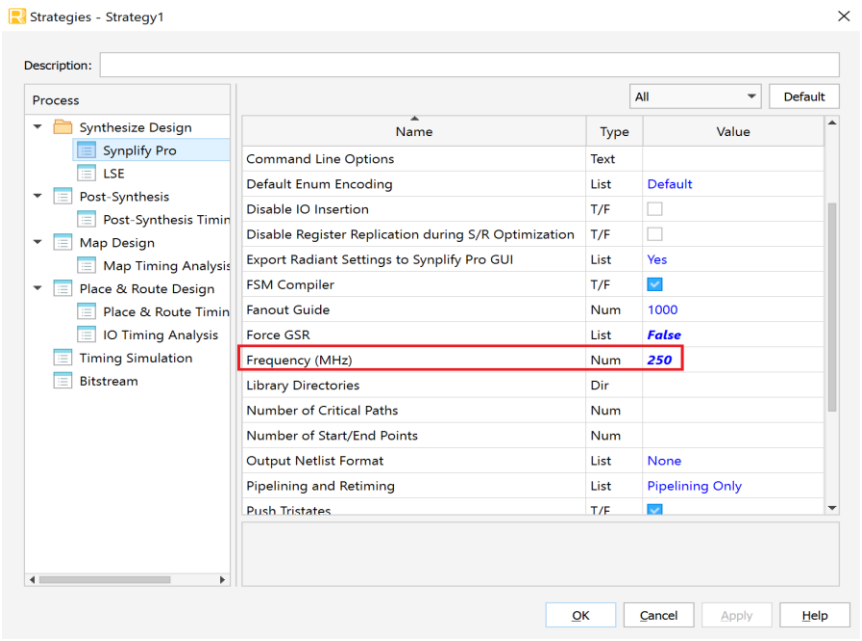

Figure C.2. Strategy for Build Generation for Main System

9. Go to the Strategy and select the Map Design
10. Select the Map Timing Analysis.
11. Select the highlighted part as mentioned in the below image.

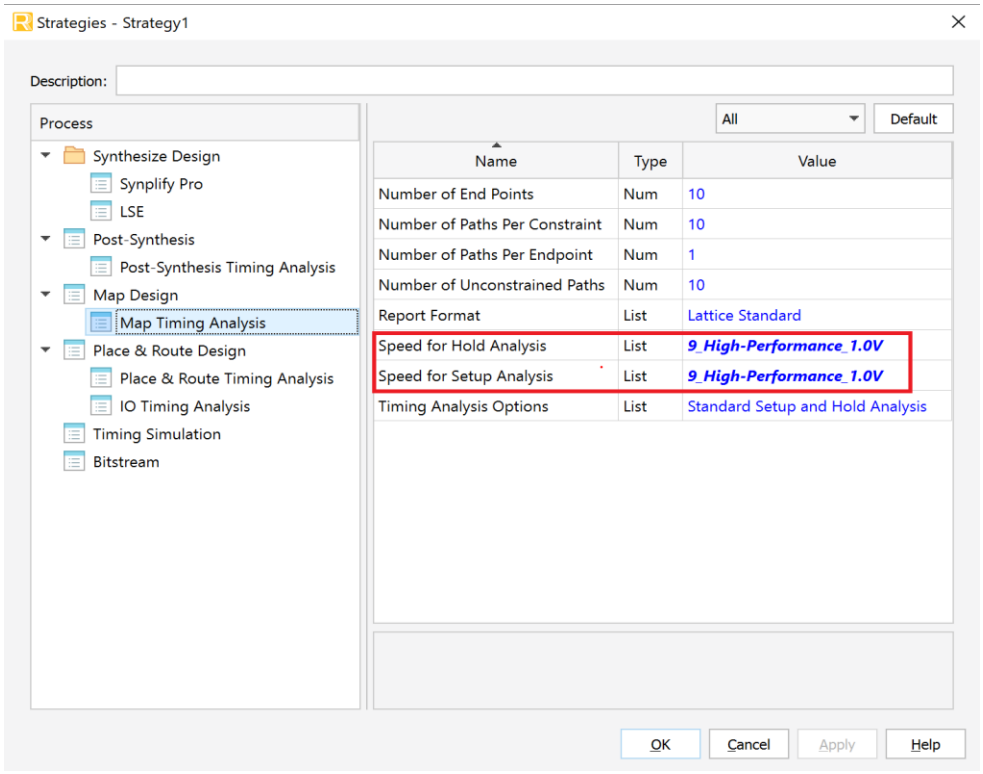

Figure C.3. MAP analysis setting for Main system bitfile generation

12. Select the Place & Route Design
13. Select the only highlighted parts as mentioned below image.

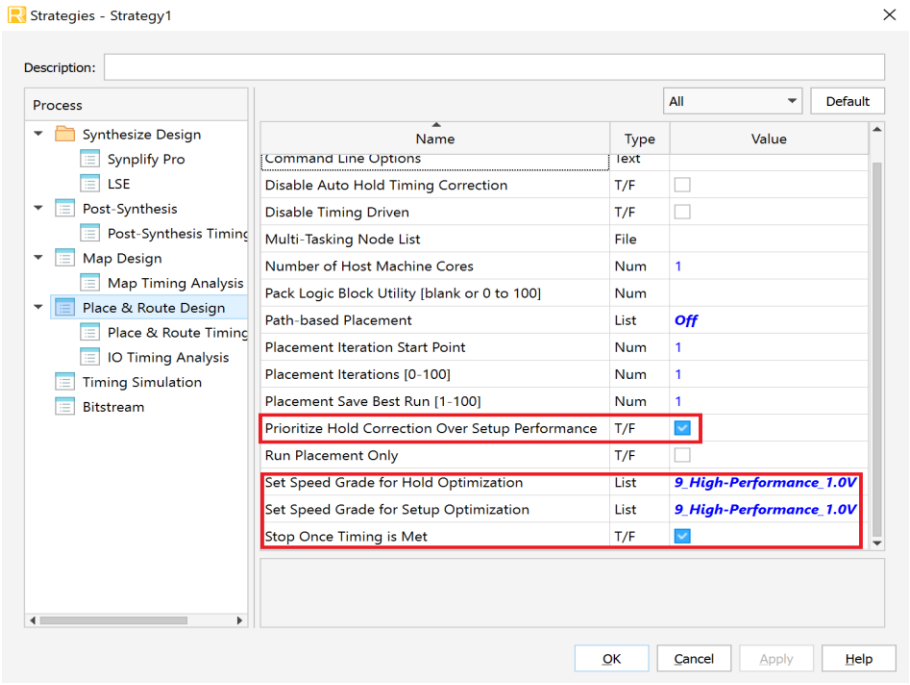

Figure C.4. PAR setting for Main system bitfile generation

14. Select the Place and Route Timing Analysis.
15. Select the only highlighted parts as mentioned in the below image.

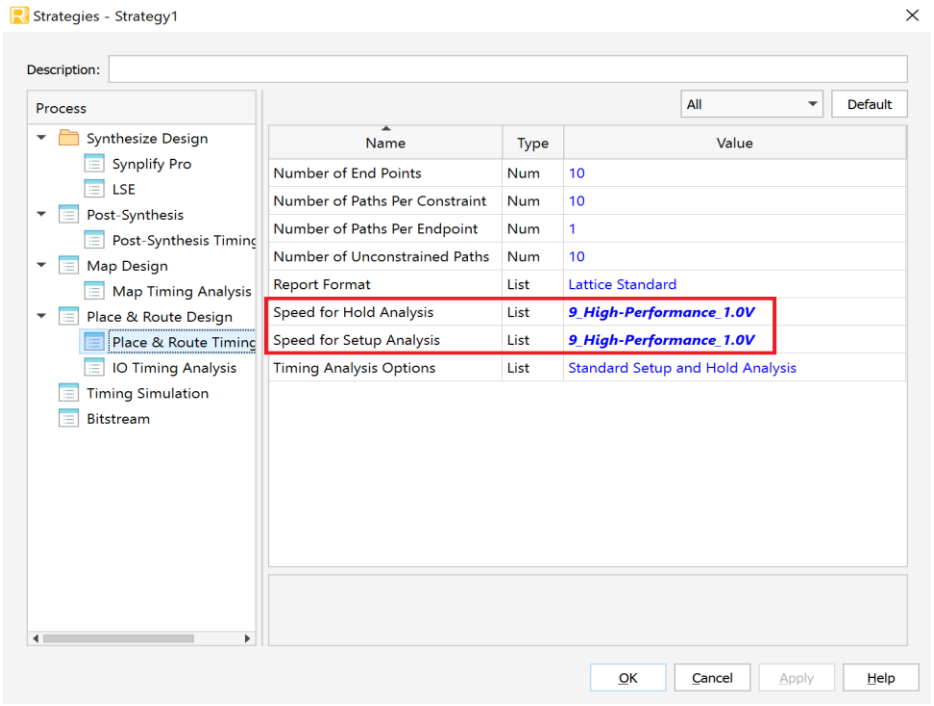

Figure C.5. PAR Timing analysis setting for Main system bitfile generation

16. Use same PDC if FPGA is same otherwise update PDC file as per FPGA pins
17. Click on the Device Constraint Editor.

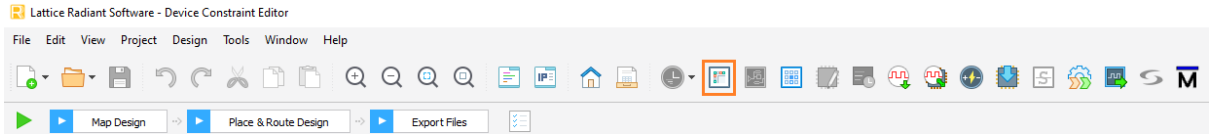

Figure C.6. Device Constraint Selection for Main System

18. Click on the Global
19. Use below Global constraint: Clock is 90 MHz

Name	Value
Junction Temperature(T)(C)	85
Voltage (V)	1
SysConfig	
SLAVE_SPI_PORT	DISABLE
MASTER_SPI_PORT	DISABLE
SLAVE_I2C_PORT	DISABLE
SLAVE_I3C_PORT	DISABLE
JTAG_PORT	ENABLE
DONE_PORT	DISABLE
INITN_PORT	DISABLE
PROGRAMN_PORT	ENABLE
BACKGROUND_RECONFIG	OFF
DONE_EX	OFF
DONE_OD	ON
MCCLK_FREQ	90
TRANSFR	OFF
CONFIG_IOSLEW	FAST

Figure C.7. Device Constraint Selection for Main System

20. Click on run all then bitfile will be generated.

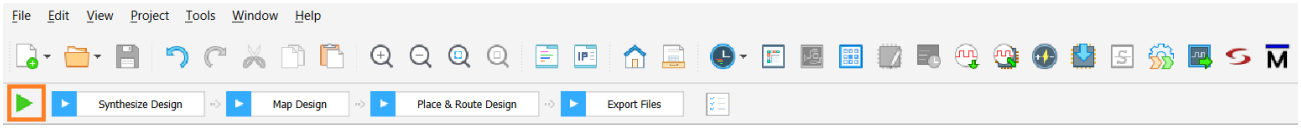

Figure C.8. Run All button

21. Open the propel builder 2.2 tool.
22. Double-click on the system0_inst.A pop-up will appear on the screen.

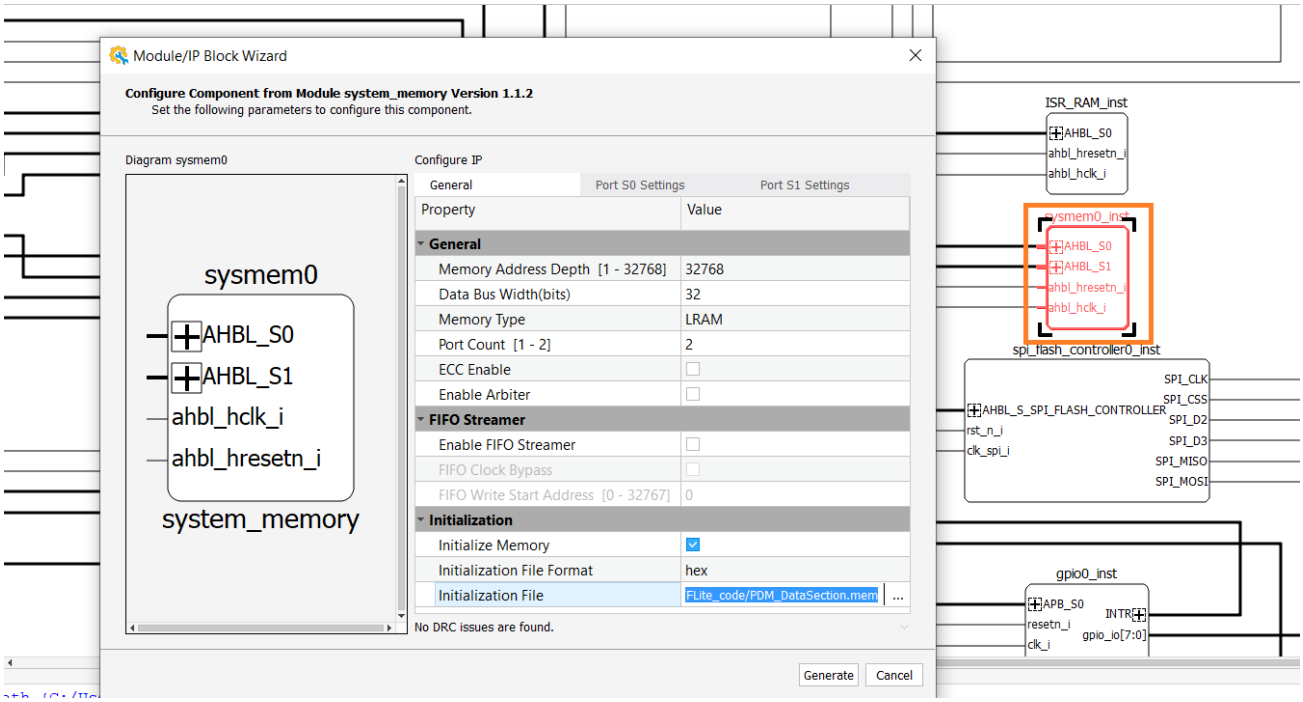

Figure C.9. system initialization file

23. Initialize Data memory with generated PDM_DataSection.mem file in TFLite_code folder of C project.
24. Click on the Generate button
25. Double-click on the ISR_RAM_inst. A pop-up will appear on the screen as mentioned in the below image.

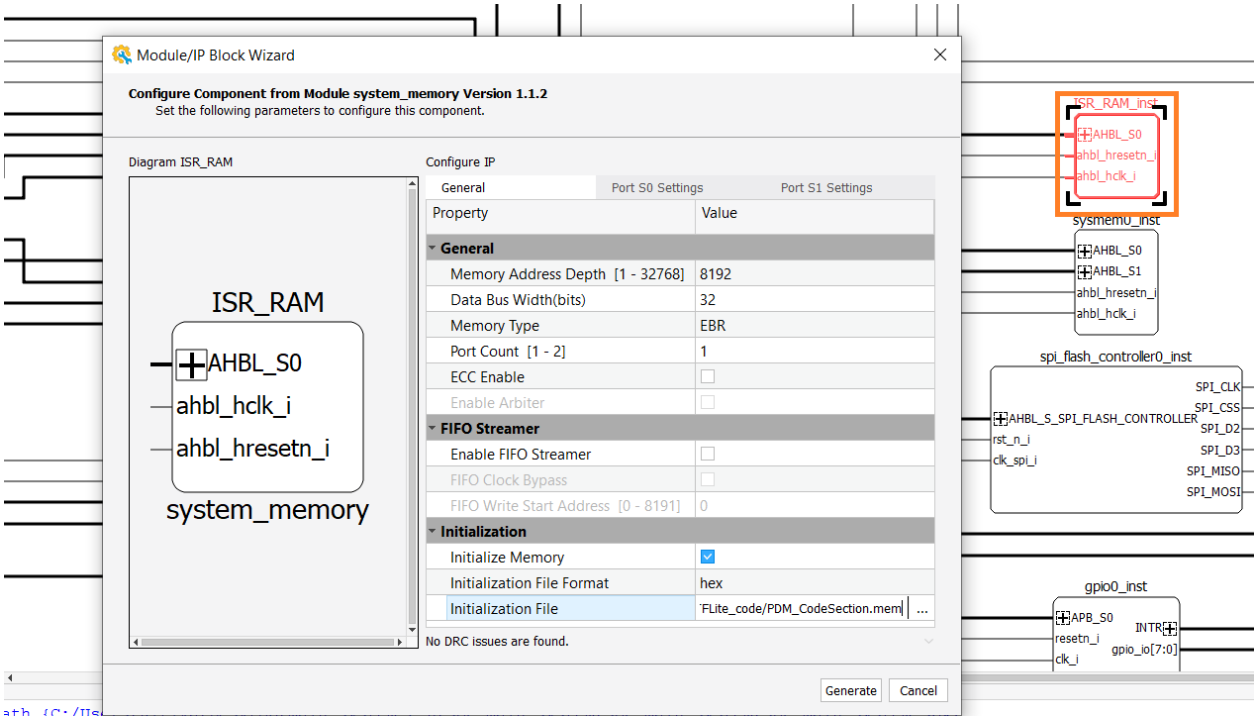

Figure C.10. ISR RAM Initialization File

26. Initialize Data memory with generated PDM_codeSection.mem file in TFLite_code folder of C project.
27. Click on the Generate button
28. Click on the Validate button

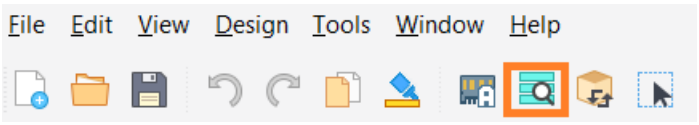

Figure C.11. Validate Button

29. Click on the Generate SGE button.

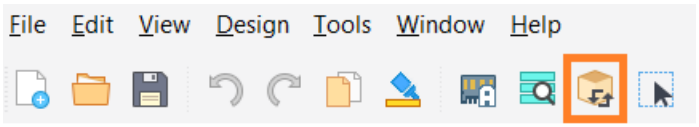

Figure C.12. Generate SGE button

30. Open the radiant tool from propel builder interface.

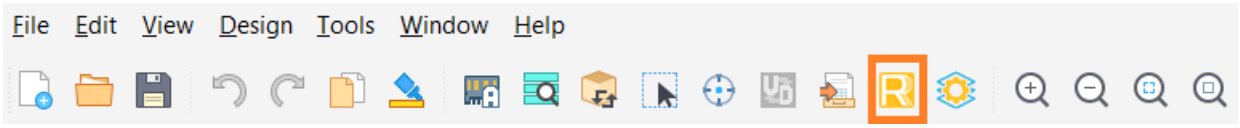

Figure C.13. Radiant tool button

31. Select FPGA and update the constraint from above points 1 to 19.
32. Click on “run all” and then bitfile will be generated.

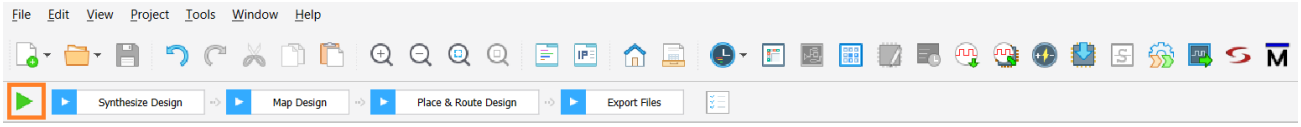

Figure C.14. Run all button

C.1.2 NODE SYSTEM

1. Open Generated Radiant Project in Radiant Tool.
2. Select Family: LFD2NX
3. Select Device: LFD2NX-40
4. Select Operating Condition: Commercial
5. Select Package: CABGA256
6. Performance Grade: 8_High-Performance_1.0V
7. Part Number: LFD2NX-40-8BG256C

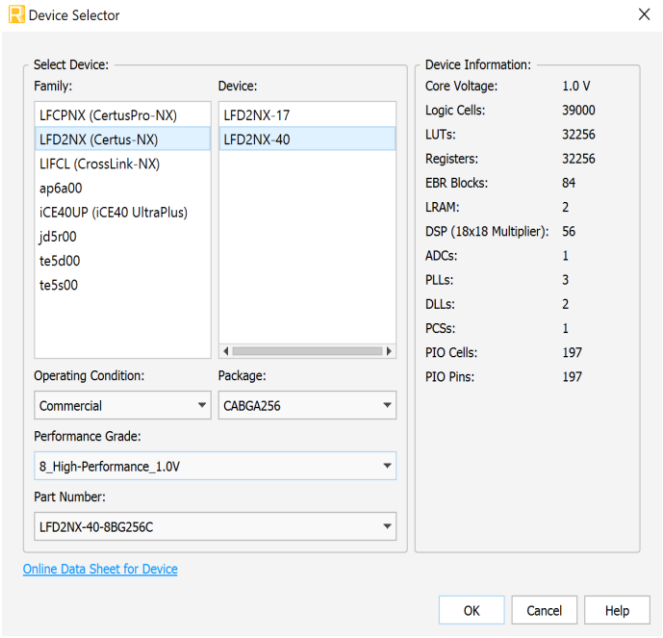

Figure C.15. Lattice Radiant Device Selector for Node System

8. Change strategy as shown below: Frequency Parameter 200 MHz

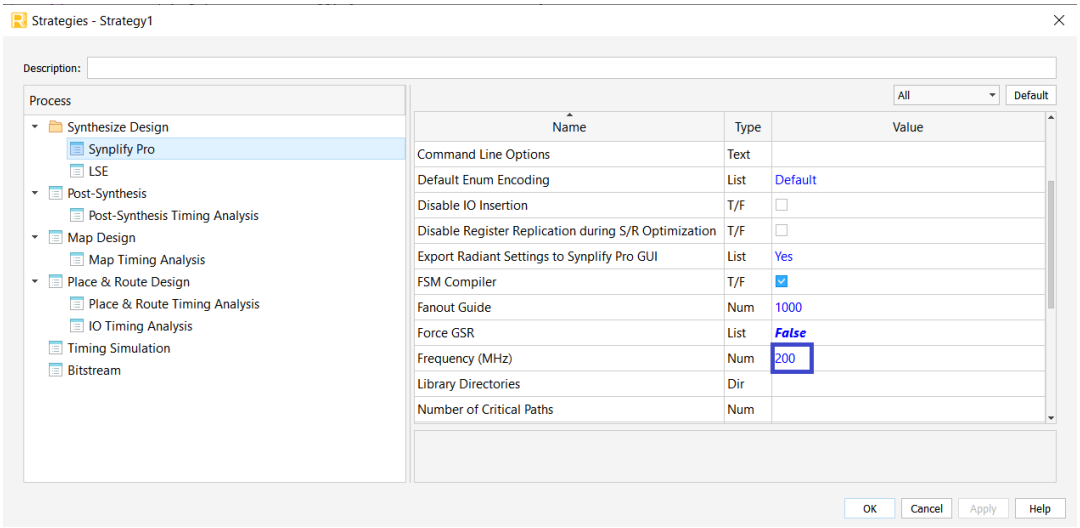

Figure C.16. Strategy for Build Generation for Node System

9. Go to the Strategy and select the Map Design
10. Select the Map Timing Analysis.
11. Select the highlighted part as mentioned in the below image.

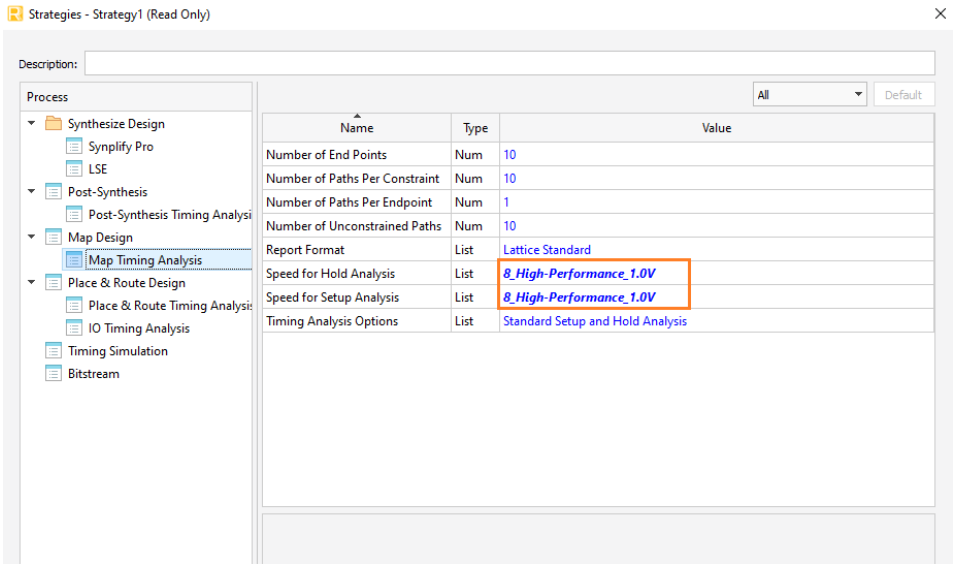

Figure C.17. MAP analysis setting for Node system bitfile generation

12. Select the Place & Route Design
13. Select the only highlighted parts as mentioned in the below image.

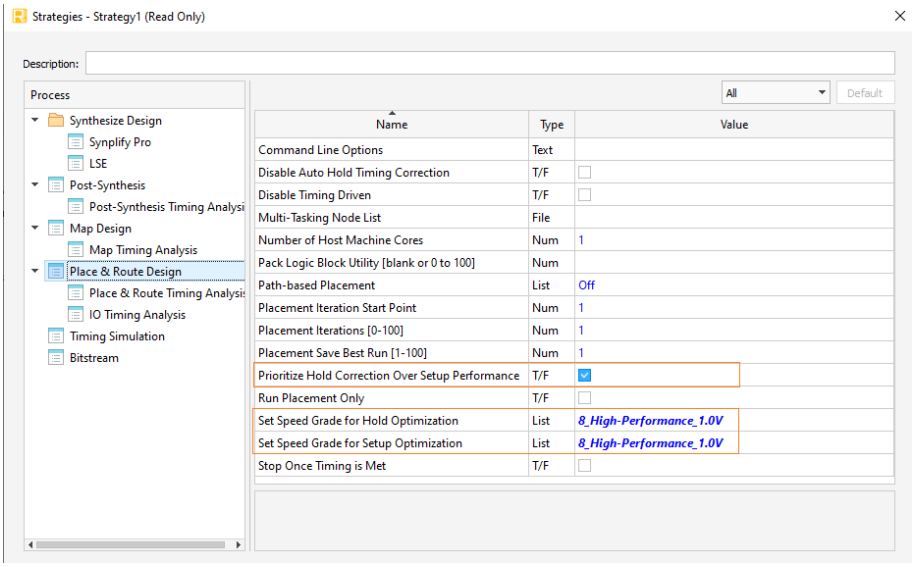

Figure C.18. PAR setting for Node system bitfile generation

14. Select the Place and Route Timing Analysis.

15. Select the only highlighted parts as mentioned in the below image.

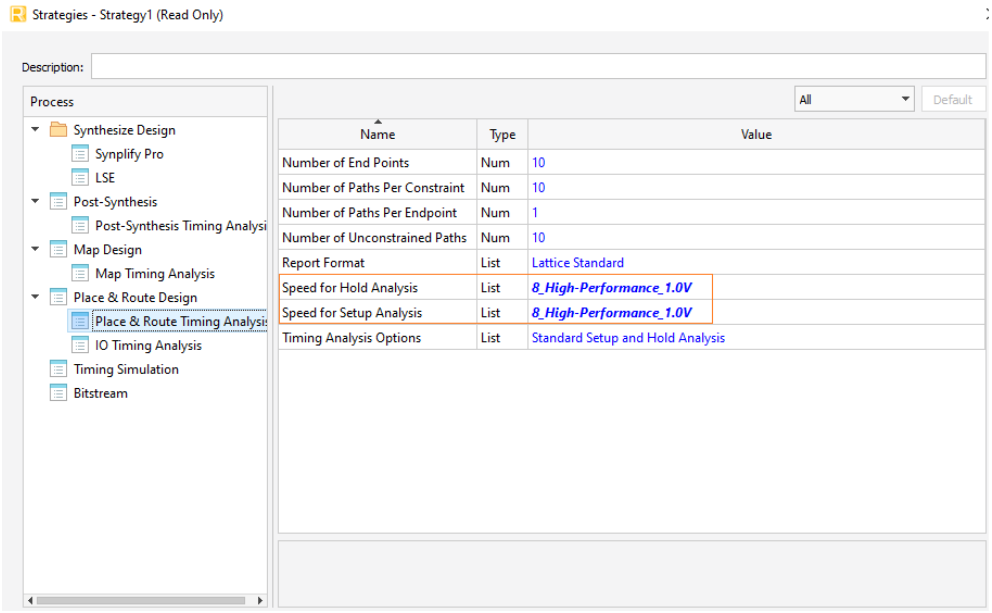

Figure C.19. PAR Timing analysis setting for Node system bitfile generation

16. Use same PDC if FPGA is same otherwise update PDC file as per FPGA pins.
17. Click on the Device Constraint Editor as mentioned below.

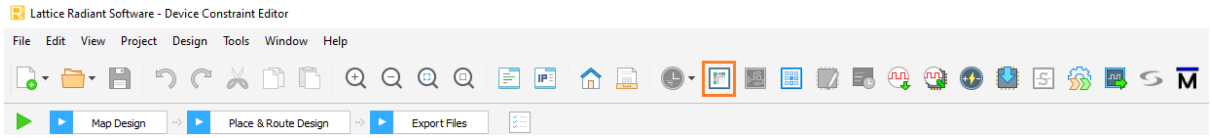

Figure C.20. Device Constraint Selection for Node System

18. Click on the Global
19. Use below Global constraint: Clock is 90 MHz.

Name	Value
Junction Temperature(T)(C)	85
Voltage (V)	0.95
SysConfig	
SLAVE_SPI_PORT	DISABLE
MASTER_SPI_PORT	DISABLE
SLAVE_I2C_PORT	DISABLE
SLAVE_I3C_PORT	DISABLE
JTAG_PORT	ENABLE
DONE_PORT	DISABLE
INITN_PORT	DISABLE
PROGRAMN_PORT	ENABLE
BACKGROUND_RECONFIG	OFF
DONE_EX	OFF
DONE_OD	ON
MCCLK_FREQ	90
TRANSFR	OFF
CONFIG_IOSLEW	FAST
CONFIG_SECURE	OFF
WAKE_UP	ENABLE_DONE_SYNC
COMPRESS_CONFIG	OFF
EARLY_IO_RELEASE	OFF
BOOTMODE	DUAL
CONFIGIO_VOLTAGE_BANK0	NOT_SPECIFIED
CONFIGIO_VOLTAGE_BANK1	NOT_SPECIFIED
User Code	
UserCode Format	Binary
UserCode	00000000000000000000000000000000
Unique Id	0000

Figure C.21. Global Constraints for Node System

20. Click on run all then bit file will be generated.

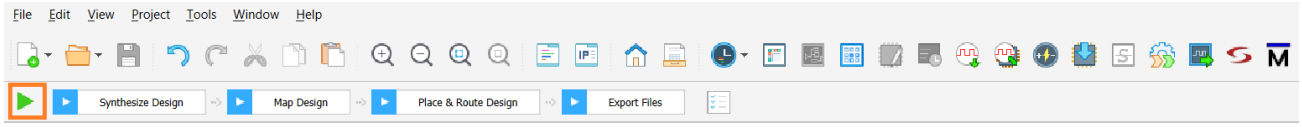

Figure C.22. Run All

21. Open the propel builder 2.2 tool.

22. Double-click on the system0_inst. A pop-up will appear on the screen as mentioned in the below image.

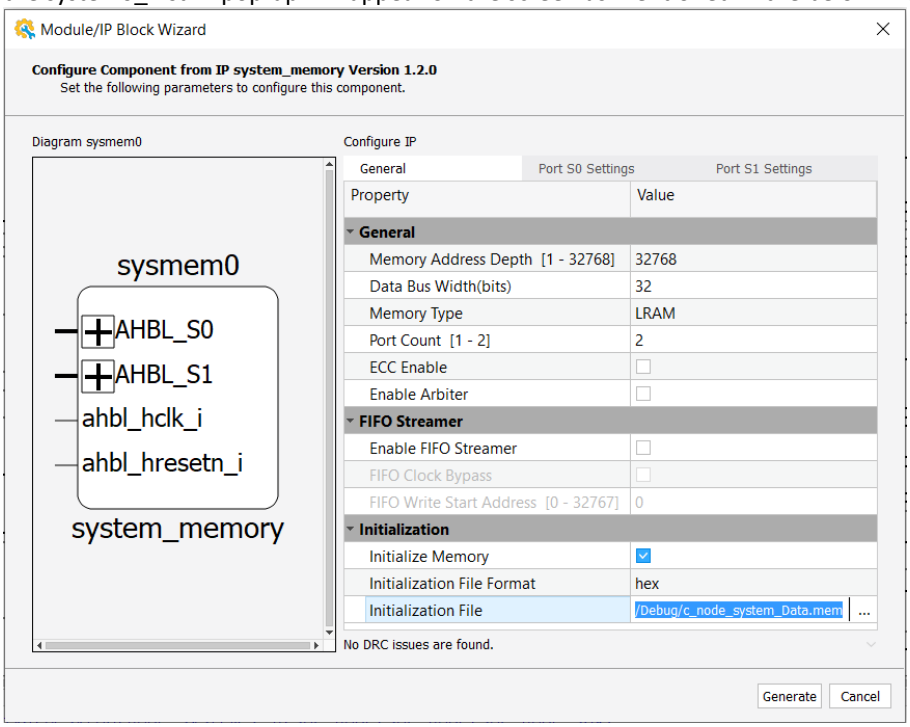

Figure C.23. system0 initialization

23. Initialize Data memory with generated c_node_system.mem file in debug folder of C project.
24. Click on the Validate button

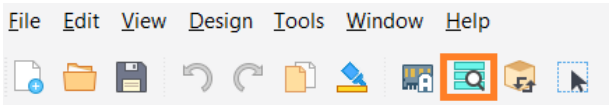

Figure C.24. Validate Button

25. Click on the Generate SGE button.

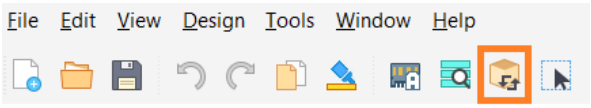

Figure C.25. Generate SGE button

26. Open the radiant tool from propel builder interface.

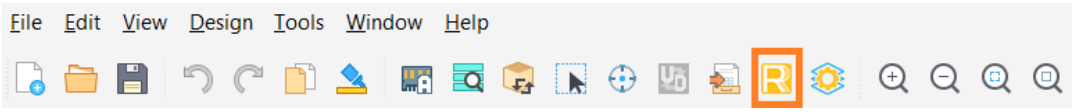

Figure C.26. Radiant Tool Button

27. Select FPGA and update constraint from above points 1 to 19.
28. Click on “run all.” A bitfile will be generated.

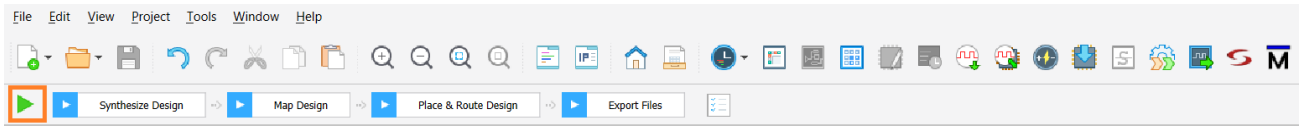

Figure C.27. Run All Button

C.2 Steps for Binary Generation

C.2.1 Main System

1. Double-click on the “Lattice Propel SDK 2022.1” to open the dialogue box as shown in fig.

Figure C.28. Propel 2022.1 application

2. To select the workspace, browse the template location

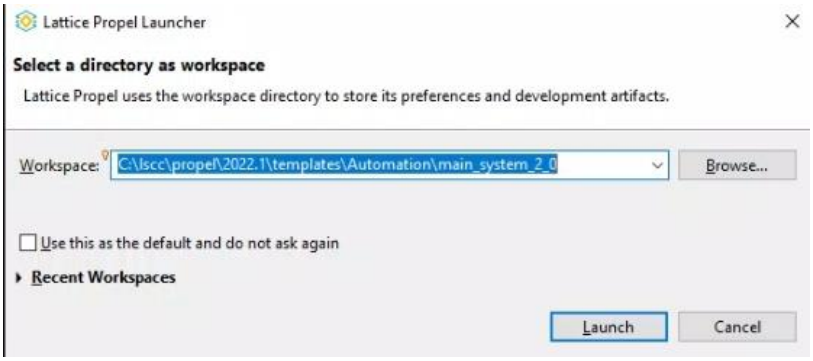

Figure C.29. Select Directory

3. “C:\lscc\propel\2022.1\templates\Automate\main_system” by clicking on the “Browse” option as shown in below image, and then click on “Launch” to launch the workspace.

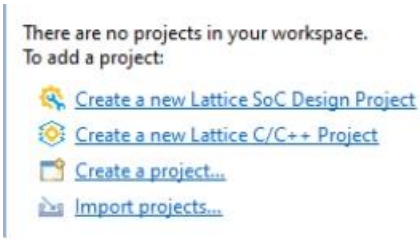

Figure C.30. Import Project

4. Click on “import” to import firmware project template.

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

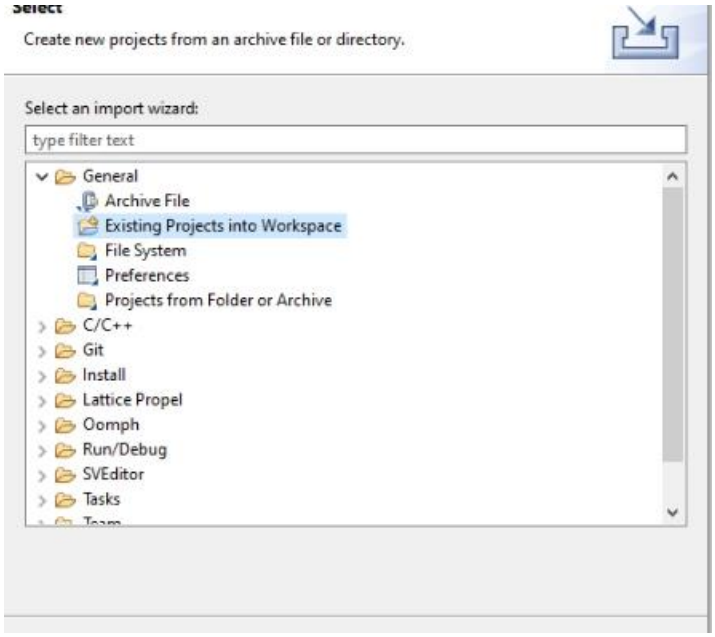

Figure C.31. Existing Project into Workspace

6. Select the root directory and browse the template location.
7. Select the project as shown in below: C:\lscc\propel\2022.1\templates\Automation\main_system_2_0
8. Click on finish.

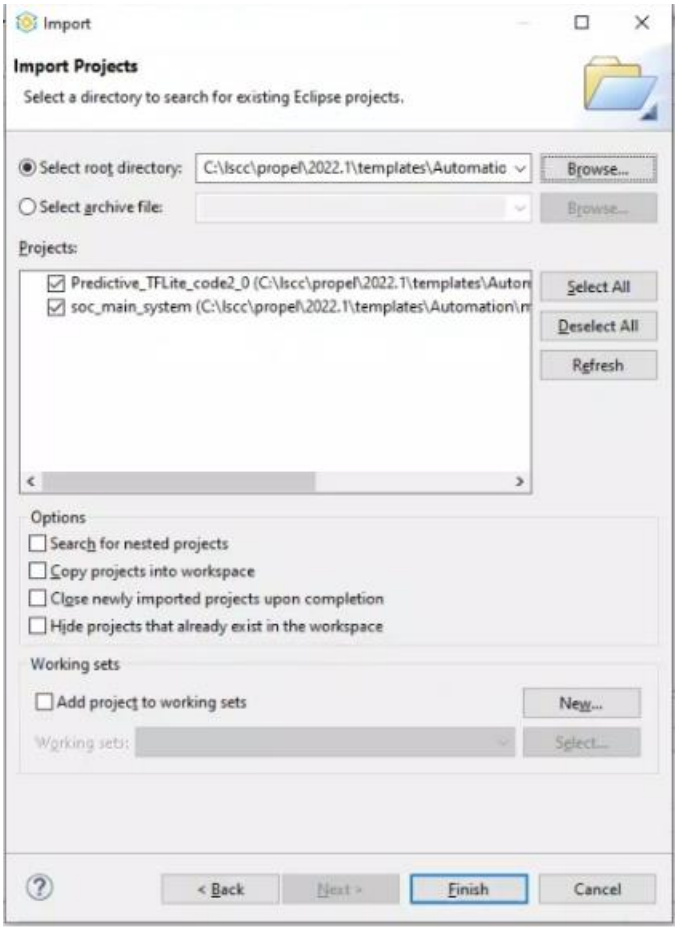

Figure C.32. Import Project

9. Right-click on the firmware project folder “Predictive_TFLite_code2_0” and select the option as shown in the below image to clean the project before building.

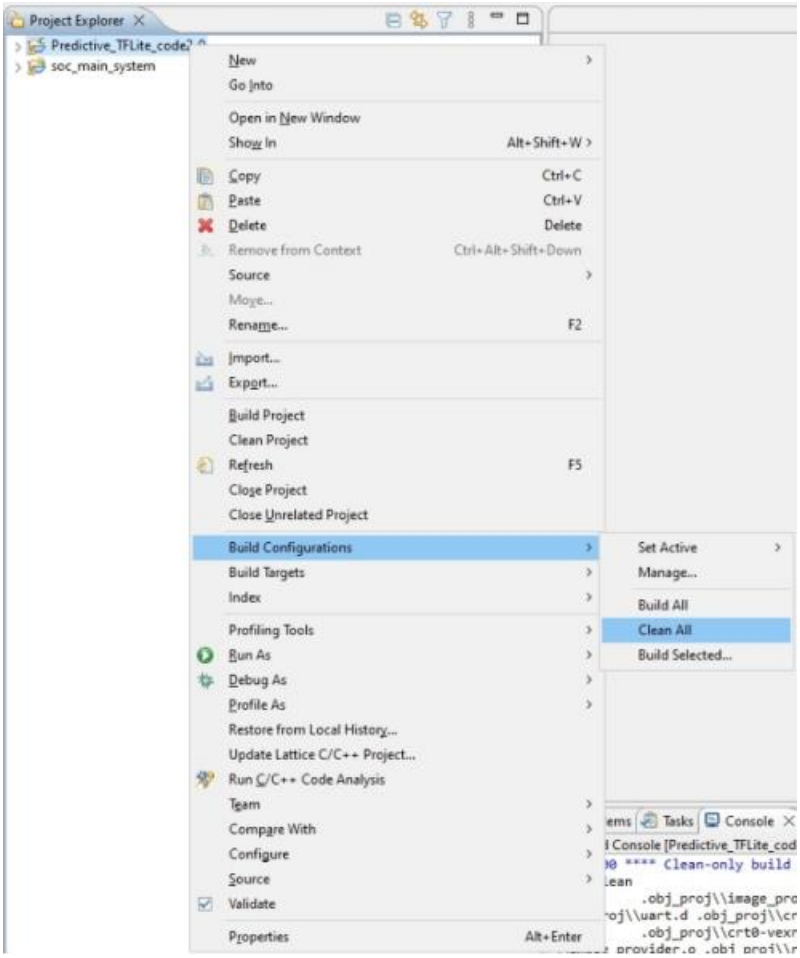

Figure C.33. Clean All Configurations

10. After selecting the option as shown in above image, observe the console and wait for the process to complete to 100%. After completion, the message shown in below image will appear on console.

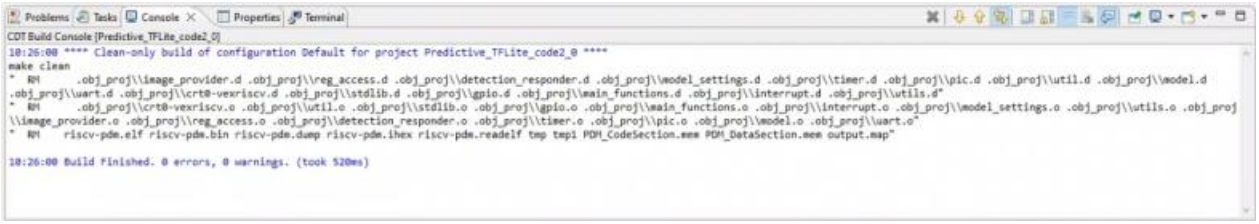

Figure C.34. Console

11. After cleaning, right-click on the “Predictive_TFLite_code2_0” and select the option as shown in below image to build the project.

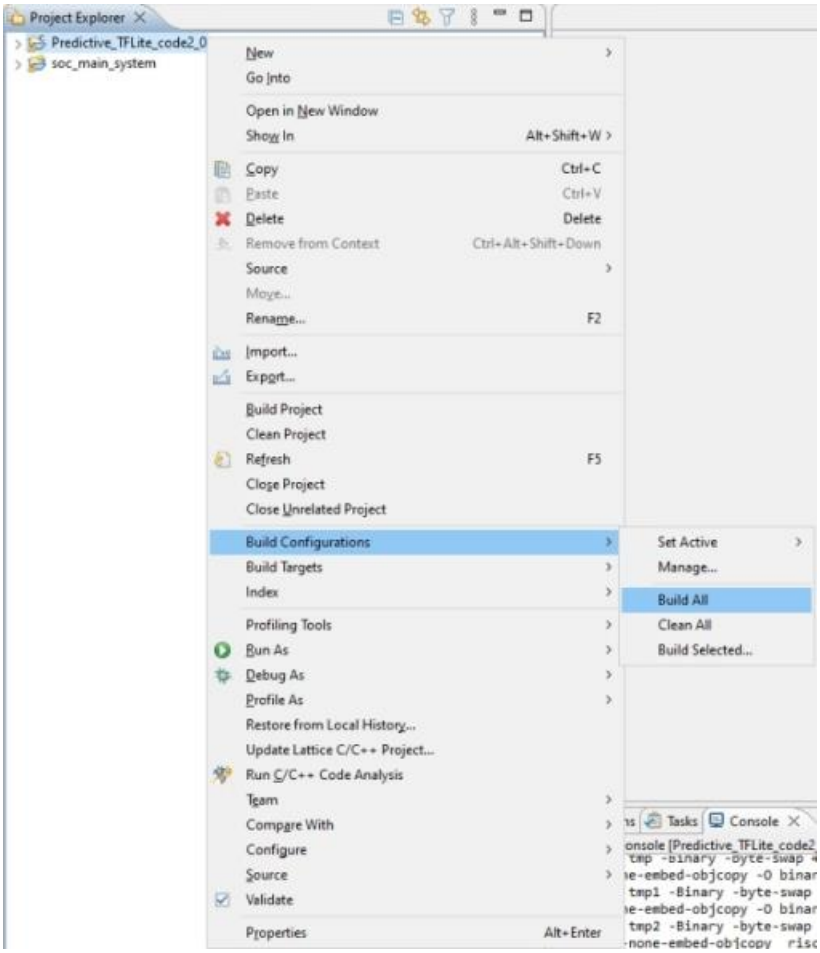

Figure C.35. Build All

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

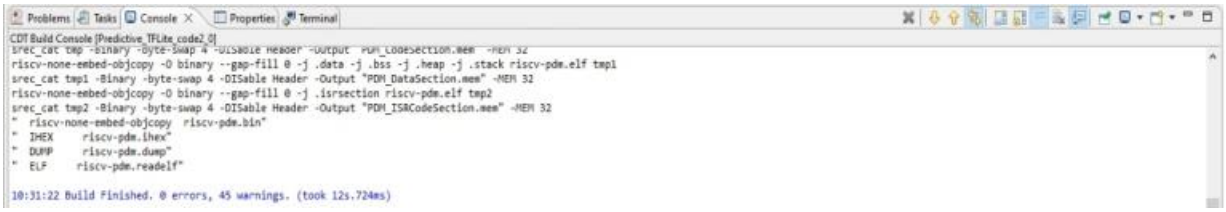

Figure C.36. Completing Process

13. To locate the binary file and .mem file to below path:
C:\lscc\propel\2022.1\templates\Automation\main_system_2_0\Predictive_TFLite_code2_0

C.2.2 Node system

1. Double-click on the “Lattice Propel SDK 2022.1” to open the dialogue box.

Figure C.37. Propel application

2. To select the workspace, browse the template location

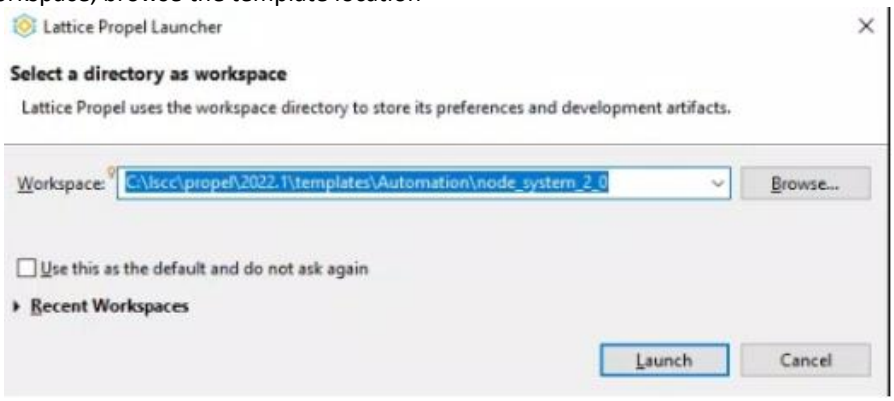

Figure C.38. Select Directory

3. “C:\lscc\propel\2022.1\templates\Automate\node_system” by clicking on the “Browse” option as shown in below image, and then click on the “Launch” to launch the workspace.

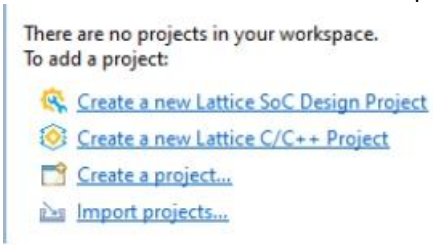

Figure C.39. Import Project

4. Click on the “import” or go to the “import” from “file” to import firmware project template.

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

Figure C.40. Existing Project into Workspace

6. Select the root directory and browse template location.
7. Select the project as shown in below: C:\lscc\propel\2022.1\templates\Automation\node_system_2_0
8. click on finish.

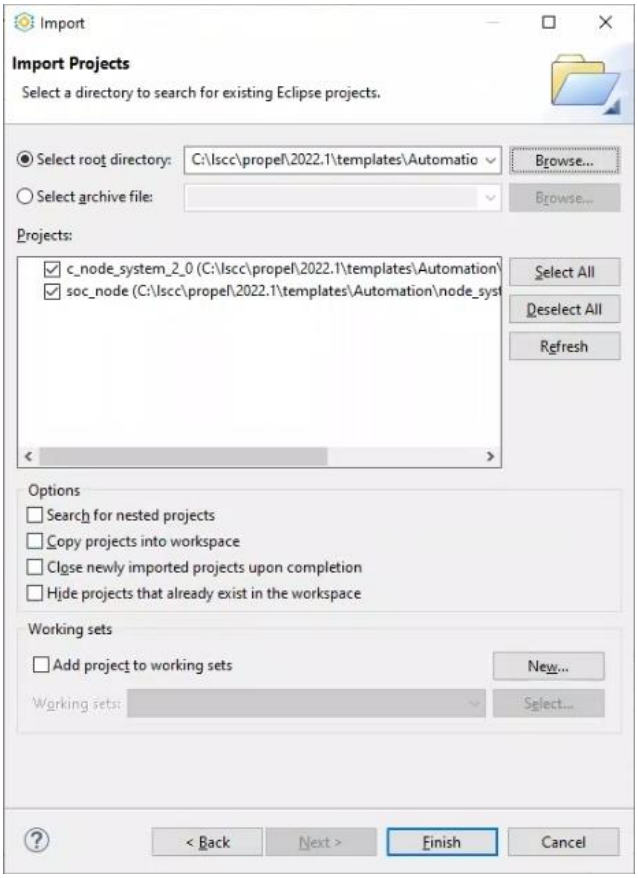

Figure C.41. Select project

9. Right-click on the firmware project folder “c_node_system_2_0” and select the option as shown in below image to clean the project before building.

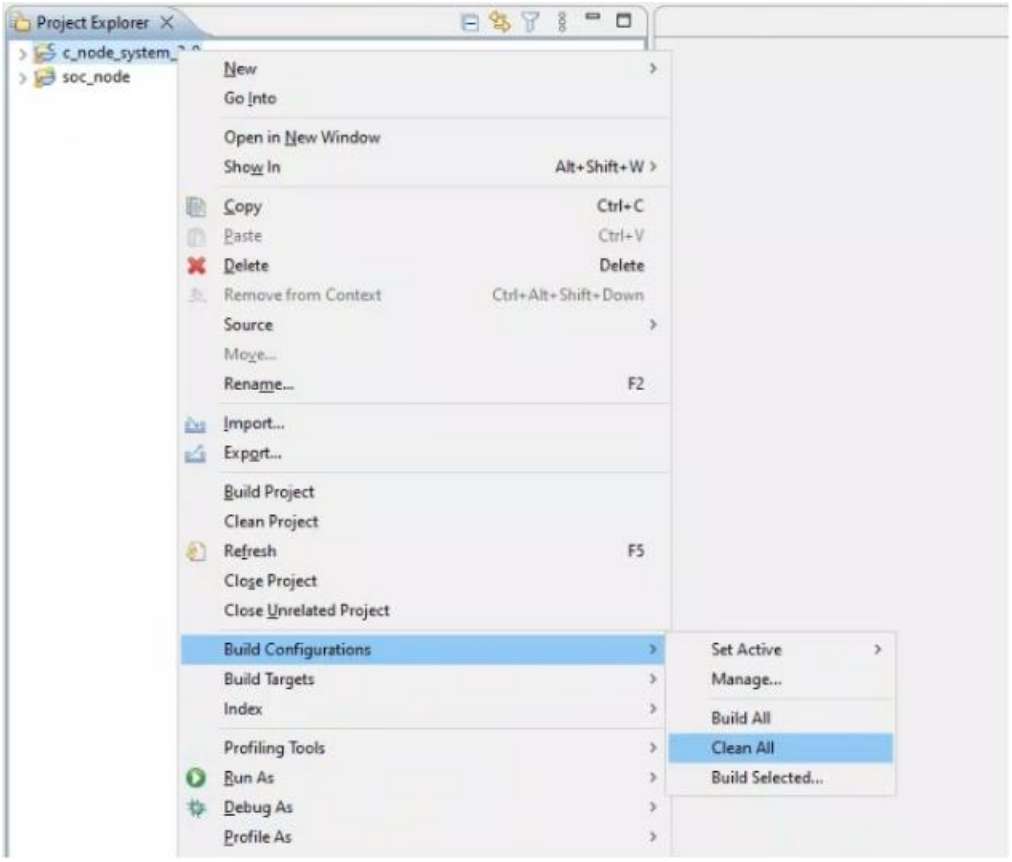

Figure C.42. Clean All Configurations

10. After selecting the option as shown in above image observe the console and wait for the process to complete to 100%. After completion, the message shown in below image appears on console.

Figure C.43. Console

11. After cleaning, right-click on the “c_node_system_2_0” and select the option as shown in below image to build the project.

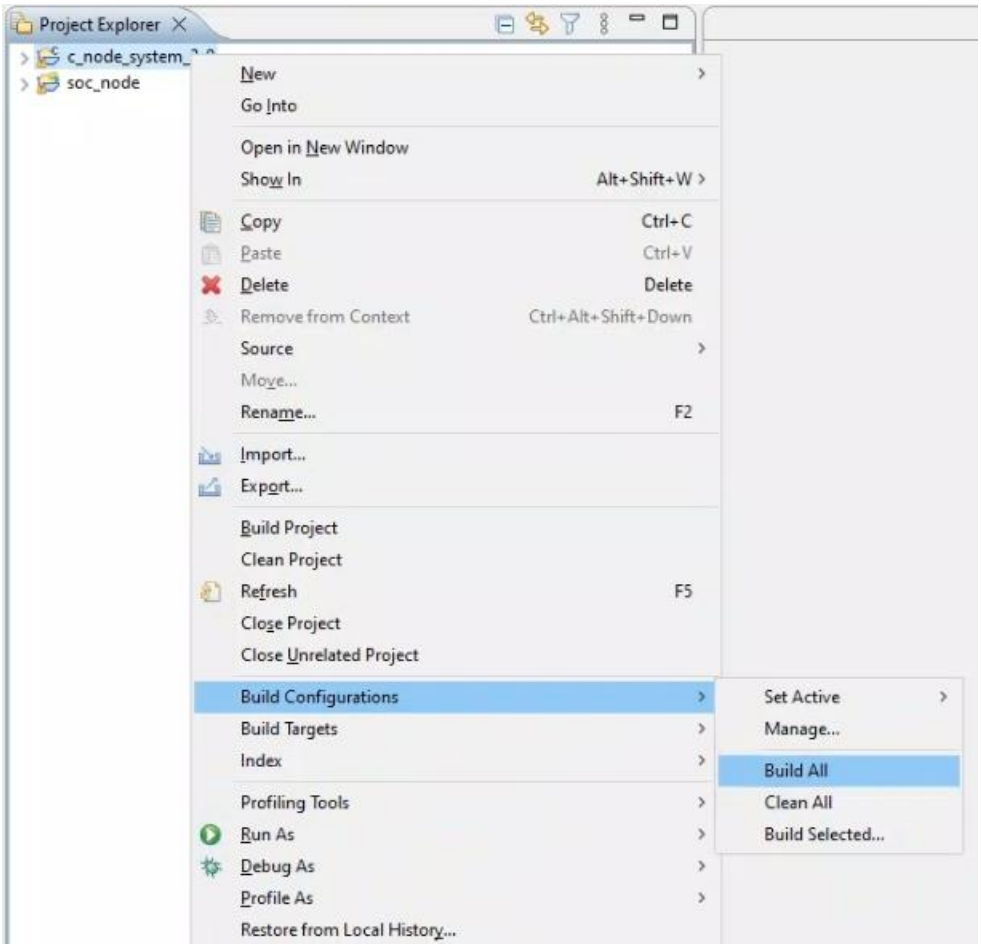

Figure C.44. Build All

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

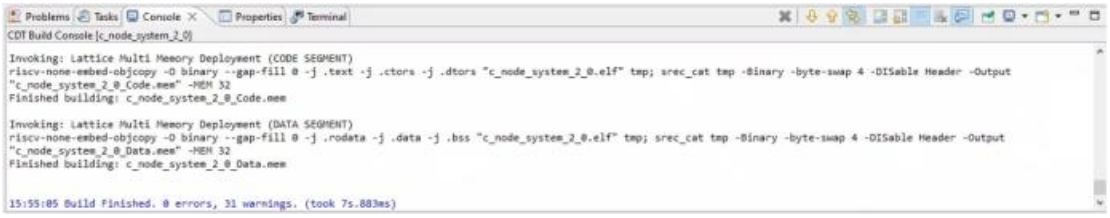

Figure C.45. Completing Process

13. To locate the binary file and .mem file to below path:
C:\lscc\propel\2022.1\templates\Automation\node_system_2_0\c_node_system_2_0

Technical Support Assistance

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

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

Revision History

Revision 1.0, March 2023

Section	Change Summary
All	Initial release.

www.latticesemi.com