POLITECNICO DI TORINO
Master Degree in Electronic Engineering

Master Degree Thesis

The Analysis and Design of Embedded Software for Ethernet Communication Protocol Standard in the Automotive sector

Supervisor
Prof. Massimo Violante

Candidate
Muhammad Haris Khan

Internship Tutor at Ideas&Motion
Ing. Marco Novaro

Academic Year 2020/202

Acknowledgments

I would like to express my deepest appreciation to all those who provided me the possibility to complete this thesis. A special gratitude I give to my professor, **Massimo Violante**, whose contribution in stimulating suggestions and encouragement, helped me to coordinate my project especially in writing this thesis.

Furthermore, I would also like to acknowledge with much appreciation the crucial role of Ideas&Motion S.r.l, who gave the permission to use all required equipment and the necessary materials to complete the task. A special thanks to my manager, **Marco Novaro**, for his understanding, patience, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis.

I also want to thank my family for the support they provided me through my entire life and in particular, I must acknowledge my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Finally, I would like to thank my friends for supporting me through the difficult yet rewarding process of writing this thesis. Their friendship made this journey more enjoyable.

Abstract

This thesis investigates the development and analysis of an Ethernet driver tailored for the S32K344 microcontroller, targeting automotive Ethernet applications. With the automotive industry's shift towards more connected and autonomous vehicles, the need for reliable, high-speed communication systems within vehicles has become increasingly critical. This research aims to fill a gap in current research by providing a detailed examination of the Ethernet protocol's implementation on the S32K344 microcontroller and developing a driver that meets the strict requirements of automotive networks.

The methodology adopted in this study involves a combination of theoretical analysis and practical development work. Initially, the Ethernet protocol, as applied to the S32K344, was thoroughly analyzed to understand its capabilities and limitations within automotive applications. This analysis informed the development phase, where a driver was created to facilitate efficient communication over Ethernet within automotive systems. Key to the development process was the use of loopback testing, which allowed for the validation of data transmission and reception without external network dependencies. The PE Micro board played a crucial role in debugging and testing the driver, ensuring its reliability and performance.

Significant findings from this work include the successful implementation of the Ethernet driver in loopback mode, demonstrating its effectiveness in handling automotive network traffic effectively. Furthermore, the driver's design and testing have laid the groundwork for future research, particularly in testing the driver across a network of multiple Electronic Control Units (ECUs) using tools like Vector CANoe. This future work aims to simulate more complex automotive network scenarios, essential for advancing the reliability and efficiency of in-vehicle communication.

The integration of the Ethernet driver within the AUTOSAR software architecture is another critical aspect of this thesis. By developing the kernel and configuration for the Ethernet driver in line with AUTOSAR standards, this work ensures compatibility with a broad range of automotive applications and highlights the importance of standardized software development in the automotive industry.

In conclusion, this thesis contributes to the field of automotive Ethernet communication by developing a robust, efficient Ethernet driver tailored for the S32K344 microcontroller. The findings not only demonstrate the driver's immediate capabilities but also pave the way for future advancements in automotive networking, emphasizing the critical role of high-performance communication systems in the next generation of automotive technology.

Table of Contents

List of Figures

1 Company Overview and Background

1.1 Company History

Ideas & Motion commenced its journey in 2013, emerging from the visionary foresight of a cadre of engineers. Over the past twenty years, this team has skillfully blended technical prowess and innovative approaches, leading to the creation of globally acclaimed automotive systems. Since its establishment in 2013, Ideas & Motion has consistently demonstrated a steadfast commitment to its mission: supporting the unique growth trajectories of its clients through innovative and tailored solutions. This mission is brought to life through two key approaches. Firstly, the company specializes in the development of bespoke smart systems, intricately designed to align with the specific needs of each customer. This personalized approach is crucial for the effective testing, validation, and assessment of new ideas and algorithms, ensuring their technical feasibility and practicality. Secondly, Ideas & Motion prides itself on its ability to rapidly transition from conceptual designs to fully-realized, engineered solutions. This swift and efficient progression is made possible by the company's reliance on a proprietary, flexible, and modular hardware and software architecture. Together, these strategies not only underline the company's dedication to its mission but also cement its role as a dynamic and innovative leader in the automotive systems industry.

1.2 Company Vision

"The company intends to support and, where required, assist the customer in the design, development and realization of a smart system in automotive, transportation, e-mobility, e-vehicle, etc. applications domain."

1.3 Company Services

Ideas & Motion has established itself as a versatile and innovative player in the automotive systems industry, offering a broad spectrum of services tailored to meet the evolving needs of the sector. These services include:

1.3.1 Engineering Services

Providing comprehensive engineering solutions that encompass the entire lifecycle of automotive system development.

1.3.2 Customized e-Motor Control

Developing specialized electronic motor control systems, tailored to enhance performance and efficiency.

1.3.3 Customized ECUs (Electronic Control Units)

Integral for modern automotive systems, tailored to client specifications.

1.3.4 Customized Embedded Software

Delivering software solutions embedded in automotive systems, designed for specific functionalities and performance criteria.

1.3.5 Virtual Prototyping

Utilizing advanced simulation techniques to create virtual prototypes, enabling efficient design and testing of automotive systems.

1.3.6 System on Chip (SoC) Design

Designing integrated circuits that consolidate all components of a computer or other electronic systems on a single chip.

1.3.7 High Performance Automotive Computational Platforms

Developing advanced computational platforms that drive high-performance automotive applications.

1.3.8 EOL (End of Line) Testing

Providing comprehensive testing services at the end of the production line to ensure product quality and performance.

2 Conventional Communication Protocols

This section discusses conventional automotive communication protocols, providing an overview of each popular protocol, which form the basis of modern vehicle communication architectures, enabling the exchange of data and control commands among electronic control units (ECUs), sensors, actuators, and other automotive components, and understanding these protocols are important to understand automotive Ethernet.

Collectively, these protocols serve a wide range of automotive communication needs, from basic functions like lighting and window control to more complex systems such as powertrain management, safety features, and infotainment systems, highlighting the critical role of communication technologies in the modern automotive industry.

2.1 CAN - Controller Area Network

The Controller Area Network (CAN) protocol is the most widely used and reliable protocol. Developed in the 1980s, it is utilized in powertrain, chassis and body domain communications for automotive applications.

The CAN design is based upon a shared bus architecture, where each node can communicate on the network without the risk of data collision utilizing an arbitration mechanism. This mechanism ensures that in the event of simultaneous communication attempts, the message with the highest priority, indicated by the lowest identifier value, is transmitted first, effectively managing access to the network.

The CAN protocol operates on a principle that combines non-return-to-zero (NRZ) bit representation with 5-bit stuffing to ensure signal integrity and error detection. The physical layer of CAN utilizes unshielded twisted pair cables (UTP), with CAN High and CAN Low lines carrying differential signals, providing resilience against electrical noise and ensuring reliable data transmission. The exact voltage levels for logic 0 and 1 depend on the implementation of the physical layer.

Traditional CAN provides a data transmission speed between 125 to 500 Kbps depending on the number of connected nodes, with the last version CAN-FD enabling transmission rates up to 2-5 Mbps.

2.2 LIN - Local Interconnect Network

LIN is specifically engineered to address the need for a cost-efficient communication network for non-critical automotive systems where high data rates and safety are not important and CAN is too expensive to implement. LIN is a serial network protocol designed to support communication between components in vehicles without requiring a microcontroller for every node in the network, significantly reducing the overall associated cost.

LIN operates at a lower data transfer rate compared to CAN, typically up to 20 Kbps, which is sufficient for the simple control tasks. This network protocol employs a single master with multiple slaves architecture, which simplifies the communication protocol and also reduces wiring complexity and cost. The master controller in a LIN network is responsible for scheduling communication and ensuring that data transmission occurs without conflict or collision, leveraging a predictable and controlled communication environment.

One of the key attributes of LIN is its support for a low-cost communication medium. It utilizes a single-wire bus, minimizing the physical infrastructure required for data transmission between nodes.

2.3 FlexRay

FlexRay is another network protocol engineered for higher data rates, reliability, and determinism. The development of FlexRay was motivated by the limitations of existing protocols like CAN and LIN in handling increasingly complex and safety-critical applications such as advanced driver-assistance systems (ADAS) and x-by-wire technologies.

FlexRay offers fault tolerance and redundancy features that are critical for safety-relevant applications and the protocol allows for dual-channel communication, where two independent channels can be used for the same data transmission, enhancing the reliability of critical message exchanges. This redundancy is essential for critical applications where failure of communication can be very hazardous. It also provides a significantly higher data transfer rate compared to CAN and LIN, with bandwidths up to 10 Mbps, which enables it to support the transmission of large volumes of data required by advanced vehicle functions.

The network architecture of FlexRay supports various topologies, including bus, star, and hybrid configurations, offering flexibility in network design to meet specific application requirements. FlexRay's design supports time-triggered and event-triggered data transmission. This hybrid approach ensures that critical messages can be transmitted at predetermined times (time-triggered) while also allowing for spontaneous message transmission when specific events occur (event-triggered).

2.4 MOST - Media Oriented Systems Transport

MOST (Media Oriented Systems Transport) protocol is specifically designed for multimedia and infotainment systems. It addresses the increasing demand for high-bandwidth, reliable multimedia data transmission, and is engineered to efficiently handle audio, video, and data streams. Since the protocol is designed to support high data rates, it enables the transmission of multimedia content at speeds up to 150 Mbps in the latest MOST150 iteration. This makes it an ideal solution for incorporating features such as satellite navigation, live traffic updates, and connectivity with mobile devices.

MOST employs a ring topology for its network architecture. In this configuration, devices are connected in a closed loop, allowing data to be transmitted in a continuous cycle through the network. This topology ensures that even if one connection in the ring is compromised, data can still be routed through the network without loss of functionality. At the core of MOST's operational efficiency is its synchronous data transmission mechanism. The protocol uses Time Division Multiple Access (TDMA) to allocate bandwidth, ensuring that each device on the network has a dedicated time slot for data transmission. This method eliminates the risk of data collision and ensures that time-sensitive multimedia content is delivered with precise timing, which is critical for maintaining audio and video synchronization and achieving high-quality playback.

2.5 Comparison with Automotive Ethernet

Features	Automotive Ethernet	CAN	LIN	MOST	FlexRay
Speed	Up to 10 Gbps	Up to 5 Mbps	Up to 20 Kbps	Up to 150 Mbps	Up to 20 Mbps
Data Handling	Packet-switched, supports large and variable size packets	Message-based, fixed length	Message-based, fixed and small length	Packet-based, optimized for audio and video	Message-based, fixed and variable length
Advantages	High bandwidth, scalability, supports advanced applications	Reliability, low cost, wide adoption	Very low cost, simple	High-quality A/V streaming, robust	Deterministic, fault tolerance, suitable for safety-critical applications
Disadvantages	Higher cost, complexity	Limited bandwidth, not suitable for data-intensive applications	Very limited bandwidth, only for simple applications	Primarily for infotainment, limited outside A/V applications	More expensive than CAN and LIN, complex setup

Table 2.1: Comparison of conventional communication protocols and automotive Ethernet

3 Automotive Ethernet

3.1 Introduction

In the rapidly advancing field of automotive technology, Ethernet has emerged as a pivotal element in the network architecture of modern vehicles. Originally designed for general data communication, Ethernet has been adapted to meet the stringent requirements of the automotive industry, offering a robust and scalable solution for in-vehicle communication. This section introduces Automotive Ethernet, highlighting its significance in enhancing vehicle functionalities, providing high-speed data transmission, and enabling features such as infotainment systems, advanced driver assistance systems (ADAS), and autonomous driving. The adoption of Automotive Ethernet marks a transformative step in the evolution of vehicle network infrastructure, promising improved performance, scalability, and flexibility.

3.2 Historical and Technical Overview

3.2.1 History of Ethernet Protocol

The Ethernet protocol, a brainchild of Xerox PARC in the 1970s, has evolved significantly from its inception. Initially designed as a local area networking technology, it quickly gained prominence due to its simplicity, efficiency, and reliability. The standardisation of

Ethernet by the IEEE 802 committee further bolstered its widespread adoption, setting the stage for its dominance in network communications.

3.2.2 The IEEE 802 Standard and Ethernet Layers

The IEEE 802 standard, a comprehensive set of networking protocols, has been instrumental in shaping Ethernet technology. This standard delineates various layers of network communication, ensuring interoperability and system robustness. Each layer, from the physical medium to the network interface, plays a pivotal role in the transmission of data, defining the operational parameters and interaction mechanisms.

3.2.3 Physical Layer (PL)

This layer is responsible for the physical transmission of data, defining the electrical and physical specifications for devices. It includes the layout of pins, voltages, line impedance, cable specifications, signal timing, and frequency. The PHY layer's primary purpose is to establish, maintain, and deactivate the physical link between communicating network systems.

3.2.4 Data Link Layer (MAC & LLC)

Divided into two sub-layers – the Logical Link Control (LLC) and the Media Access Control (MAC), the data link layer is responsible for node-to-node data transfer and error detection and correction. The MAC sub-layer manages protocol access to the physical network medium, while the LLC provides flow and error control.

- Media Access Control (MAC): MAC is a sub-layer that controls protocol access to the physical network medium. It addresses devices uniquely at the hardware level and facilitates multiple devices to communicate within a network. MAC addresses are unique 6-byte codes assigned to every device on an Ethernet network.

- Logical Link Control (LLC): This sub-layer manages communication between devices over a single link of a network. LLC provides multiplexing mechanisms that allow different network protocols to coexist within a multipoint network and offers flow control, error checking, and synchronization.

3.2.5 Network and Transport Layers

Although not defined explicitly in the IEEE 802 standard for Ethernet, these layers are crucial in broader networking contexts. They manage data routing, delivery, and integrity across complex networks.

3.2.6 Ethernet Frame Structure

The Ethernet frame is the fundamental unit of data transmission in Ethernet networks. It consists of several fields, each serving a specific purpose in the communication process. The frame starts with a preamble and SFD (Start Frame Delimiter), followed by destination and source MAC addresses, type/length field, payload, and ends with a Frame Check Sequence (FCS) for error checking.

- **Preamble and Start Frame Delimiter (SFD):** The frame starts with an 8-byte preamble (7 bytes) and SFD (1 byte). The preamble helps in synchronizing the receiver before actual data is sent.

- **Destination MAC Address:** This 6-byte field specifies the recipient of the frame.

- **Source MAC Address:** This 6-byte field specifies the sender of the frame.

- **Type/Length:** A 2-byte field indicating the type of protocol or the length of the payload.

- **Payload/Data:** The actual data being transmitted, which varies in size.

- **Pad:** It guarantees a minimum length.

- **Frame Check Sequence (FCS):** A 4-byte field used for error checking. The sender calculates the FCS based on the frame's data and adds it to the frame. The receiver recalculates it for error detection.

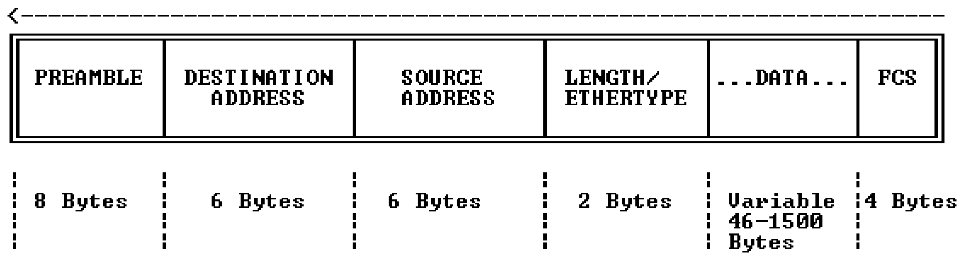


Figure 3.1: Ethernet II Frame

4 Software Tools

4.1 Visual Studio

Visual Studio is an advanced integrated development environment (IDE) developed by Microsoft, primarily known for its efficiency and versatility in various programming domains, including embedded C programming. It stands out as a popular choice among developers for creating, testing, and deploying embedded systems.

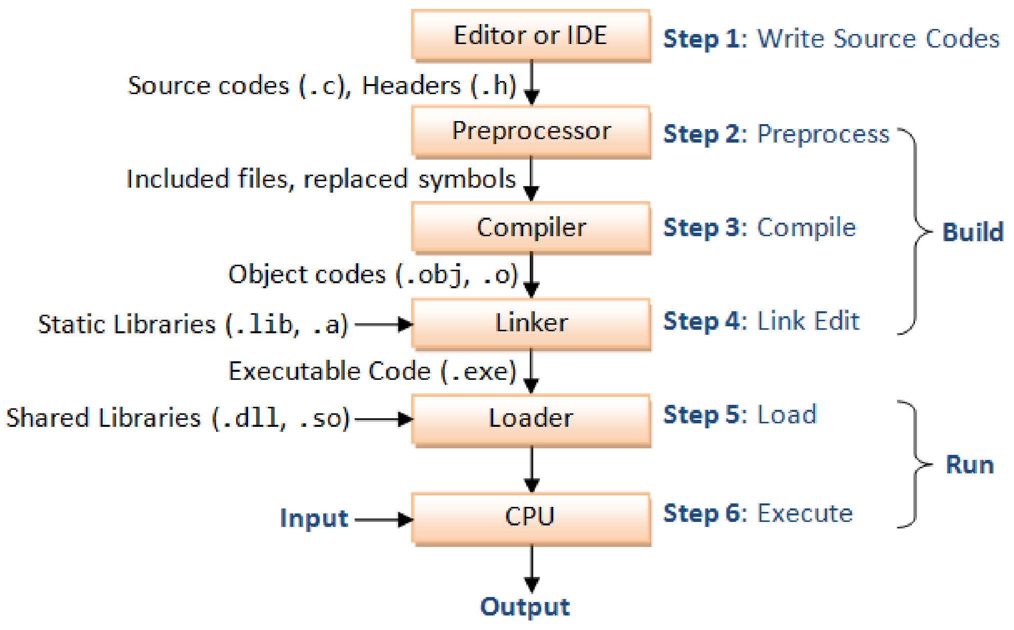


Figure 4.1: VS Code Flow

4.1.1 Key Features of Visual Studio in Embedded C Programming

Comprehensive Integrated Development Environment: Visual Studio integrates essential tools like a code editor, debugger, and project management utilities into one platform, streamlining the development process for embedded systems.

4.1.2 Advanced Debugging Tools

The IDE excels with its sophisticated debugging features, which are crucial for the complex troubleshooting required in embedded C programming.

4.1.3 IntelliSense and Code Analysis

With IntelliSense for smart code completions and powerful code analysis tools, Visual Studio aids in writing efficient and error-free code, a critical aspect in resource-constrained embedded environments.

4.1.4 Cross-platform Development Capabilities

Its support for cross-platform development enables programmers to create code compatible with various hardware and operating systems, a common necessity in the embedded systems landscape.

4.1.5 Rich Libraries and APIs Access

The IDE offers extensive libraries and APIs, simplifying many tasks in embedded programming and providing robust solutions for common challenges.

4.1.6 Vibrant Community and Consistent Support

A strong community and continuous support from Microsoft ensure that developers have access to help, new trends, and regular updates, keeping the tool relevant and effective.

4.1.7 Customizability with Extensions

The ability to customize Visual Studio with various plugins and extensions allows developers to tailor the environment to the specific needs of their embedded projects.

4.2 S32 Design Studio IDE: An In-depth Overview

S32 Design Studio IDE stands as a comprehensive development environment tailored specifically for the demands of automotive applications and high-reliability microcontrollers and processors. This robust software platform offers extensive support across multiple operating systems including Windows, Ubuntu, Debian, and CentOS. It is made available to users at no financial cost, embodying an accessible solution for developers worldwide.

4.2.1 Cross-Platform Compatibility

One of the key features of S32 Design Studio IDE is its cross-platform compatibility. This flexibility allows developers to seamlessly operate across various systems, catering to diverse development environments and requirements.

4.2.2 Comprehensive Development Toolkit

At its core, S32 Design Studio IDE amalgamates essential development tools within a single package. It incorporates the functionality of the Eclipse IDE for editing, the GCC for compiling, and the GDB for debugging, streamlining the development workflow. This integration facilitates a cohesive and efficient development experience, from writing code to troubleshooting and optimization.

4.2.3 No-Cost Accessibility

The IDE's free availability stands as a testament to its commitment to democratizing development resources. By removing financial barriers, it ensures that developers and organizations, regardless of size or budget, can access high-quality development tools. This approach fosters innovation and development within the automotive and microcontroller sectors.

4.2.4 Integration with NXP's Ecosystem

Enhancing its utility, S32 Design Studio IDE is intricately integrated with NXP's suite of tools and software. This integration provides a seamless development experience, enabling users to leverage NXP's advanced technologies and solutions efficiently. It supports a broad spectrum of NXP products, accommodating various architectures and facilitating versatile project development.

4.2.5 Embedded Software Development

Designed with embedded software in mind, the IDE supports the S32 Software Development Kit (SDK), FreeRTOS, among other tools, offering a robust foundation for embedded software creation. This comprehensive support ensures developers have access to the necessary resources to develop, test, and deploy embedded systems efficiently.

4.2.6 Compatibility with Multiple Compilers and Debuggers

Acknowledging the diversity in development preferences, S32 Design Studio IDE is compatible with a variety of compilers and debuggers. It supports NXP GCC, Green Hills, IAR compilers, and features a GDB interface for debugging. This compatibility allows developers to choose the tools that best fit their project needs and preferences.

4.2.7 Specialized Tools for Advanced Projects

For projects with specific requirements, such as vision and radar applications, the IDE includes specialized tools like the NXP APU Compiler. These tools address the unique needs of advanced projects, enabling developers to tackle complex tasks with precision and efficiency.

4.3 Wire shark

Wireshark is a highly versatile and powerful network packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education.

4.3.1 Key Details

4.3.2 History

Wireshark was initially released around 1998 by Gerald Combs as "Ethereal." When Combs joined CACE Technologies in 2006, he renamed the project Wireshark due to trademark issues. Over the years, Wireshark has been supported and sponsored by various organizations, with Sysdig becoming its primary sponsor in 2022 and establishing the Wireshark Foundation in 2023

4.3.3 Functionality

Wireshark captures and analyzes network packets in real time. It can dissect and display the contents of packets, with detailed information about each packet's structure and content. Wireshark supports a wide range of network protocols and can read packets from various types of networks, including Ethernet, IEEE 802.11 (Wi-Fi), PPP, and loopback networks.

4.3.4 Data Analysis

It allows users to browse captured network data using a graphical user interface or via a command-line utility called TShark. Users can apply filters to focus on specific types of traffic, and it's possible to edit or convert captured files programmatically.

4.3.5 VoIP Analysis and More

Wireshark can detect VoIP calls in the traffic and, in some cases, play the media flow. It also supports capturing raw USB traffic and wireless connections.

4.3.6 Compatibility

Wireshark uses pcap (packet capture) to capture packets, meaning it can capture packets on networks supported by pcap. Its native file formats are the libpcap format and pcapng format, which are compatible with other network analyzers and tools.

4.3.7 Security Considerations

Capturing raw network traffic can require elevated privileges on some platforms. Earlier versions of Wireshark often ran with superuser privileges, but newer versions use dumpcap for traffic capture, reducing the need for elevated privileges.

4.3.8 Color Coding

To assist in analysis, Wireshark can color-code packets based on user-defined rules, making it easier to identify different types of traffic.

5 Hardware

5.1 S32K344 EVB

The S32K344 EVB is an evaluation and development board designed for general-purpose industrial and automotive applications. It's based on the 32-bit Arm Cortex®-M7 S32K3 MCU in a 172 MaxQFP package.

5.1.1 Core

The board features the Cortex-M7 core, which is a high-performance processor designed for a variety of applications.

5.1.2 Debug Interface

It supports JTAG/SW (Serial Wire) for debugging, which is a standard method for connecting a debugger to a device.

5.1.3 Safety and Security Features

It offers dual cores configured in lockstep mode, ASIL D safety hardware, and HSE security engine. ASIL D is the highest level of automotive safety integrity, and HSE (High Security Engine) indicates advanced security features.

5.1.4 Connectivity and Power

The board supports advanced connectivity options and is designed for low power consumption, which is crucial for automotive and industrial applications.

5.1.5 Over-The-Air (OTA) Support

This feature allows for remote updating of the software running on the device, which is particularly useful in automotive applications where direct physical access to the control units can be challenging.

5.1.6 Expansion Options

The board is compatible with the Arduino® UNO pin layout, allowing for a broad range of expansion board options. This makes it suitable for quick application prototyping and demonstrations.

5.1.7 CMSIS Drivers

It includes a CMSIS-Driver for the VIO interface, an abstraction for peripherals typically used in example projects. For instance, the Blinky example uses this interface to create a blinking light with the USER LED on the board.

5.1.8 Key Ethernet Features

The chip we are examining exhibits versatile Ethernet capabilities, primarily through its support for both MII and RMII interfaces. Below are the salient features of its Ethernet functionalities:

5.1.8.1 Interface Support and Performance

- The chip is equipped with a 4-bit Media Independent Interface (MII), capable of operating at frequencies of 2.5, 25, or 50 MHz. This allows the MII to facilitate data transmission speeds of 10, 100, or 200 Mbps.
- It also includes a 2-bit Reduced Media Independent Interface (RMII), operating at a steady frequency of 50 MHz, and capable of handling speeds of 10 or 100 Mbps.
- Additionally, the chip features a 4-bit MII-Lite interface, designed to function at lower frequencies of 2.5 or 25 MHz.

5.1.8.2 Precision Timing and Data Rate

- The chip supports four Precision Time Protocol (PTP) outputs, which can be effectively utilised through Direct Memory Access (DMA) triggering.
- It boasts a substantial data rate capacity, supporting up to 200 Mbps on the Ethernet MII interface.

5.1.8.3 Memory and Storage Capacities

- The chip is designed with a sizable Media Transfer Layer (MTL) Receive FIFO (First In, First Out), which has a capacity of 8192 bytes.
- Similarly, the MTL Transmit FIFO also holds 8192 bytes, ensuring efficient data handling during transmission processes.

These features collectively enhance the chip's Ethernet performance, making it suitable for various applications that require reliable and high-speed data communication.

5.1.9 Usage

The S32K344 EVB is used primarily in industrial and automotive applications, where high performance, safety, security, and low power consumption are critical. It is used for developing, testing, and prototyping applications before they are deployed in actual automotive or industrial systems.

This board to write and test software that will run on the S32K3 microcontrollers. It can be used to simulate real-world scenarios and ensure that software behaves as expected in various conditions.

In summary, the S32K344 EVB is a versatile, high-performance development board suited for demanding applications in the automotive and industrial sectors, providing developers with a platform for creating, testing, and prototyping sophisticated applications.

5.2 PE Micro Debugger

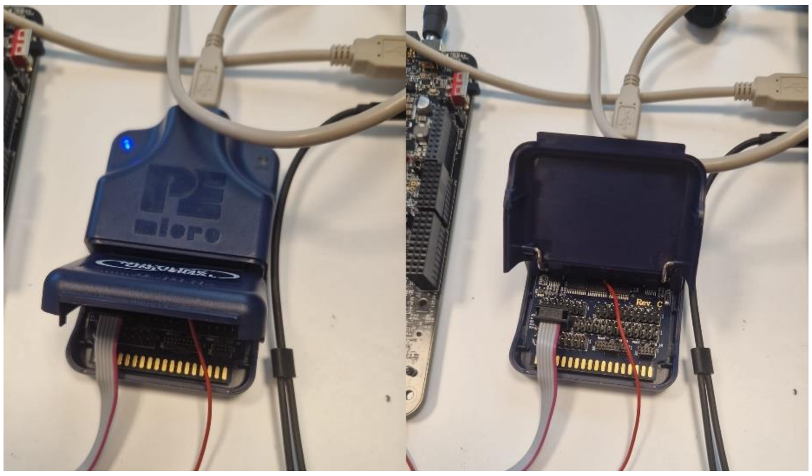


Figure 5.1: PE Micro Hardware Debugger

PEmicro, or P&E Microcomputer Systems, is known for its embedded tools for flash programming and development. They offer a range of tools and solutions for different development and debugging needs.

5.2.1 Development and Debugging Tools

PEmicro provides tools that support both C-level and assembly-level development. This includes debugger software, hardware interfaces, and complete packages that integrate additional software tools into a cohesive development environment. For high-level devices, such as ColdFire and Power PC Nexus, their full C-level development packages include the GNU C compiler.

5.2.2 Hardware Interfaces

They offer hardware interfaces for a variety of project needs, whether for cost-sensitive projects or those requiring increased flexibility. These interfaces can transition seamlessly to production, making them versatile for different stages of product development.

5.2.3 Flash Programming Solutions

PEmicro also caters to users who need to program the FLASH memory of their target system. This is particularly useful for those who already have a debugger and need a fast and affordable programming solution. Their solutions include both the 'PROG' line of flash programming software and hardware solutions like Cyclone and Multilink.

5.2.4 Custom PC Applications

PEmicro has created libraries (like DLLs for Windows and SOs for Linux) that allow users to access the low-level functionality of PEmicro hardware interfaces. This feature is useful for custom programming solutions, test and verification procedures, or any application that requires interfacing with a microcontroller target from a PC.

PEmicro debuggers are used in various stages of product development, from initial design and development to production. They are essential tools for developers working with embedded systems, offering a range of solutions for programming, debugging, and interfacing with microcontrollers and other embedded devices.

6 Ethernet Media Access Controller (EMAC) Architecture

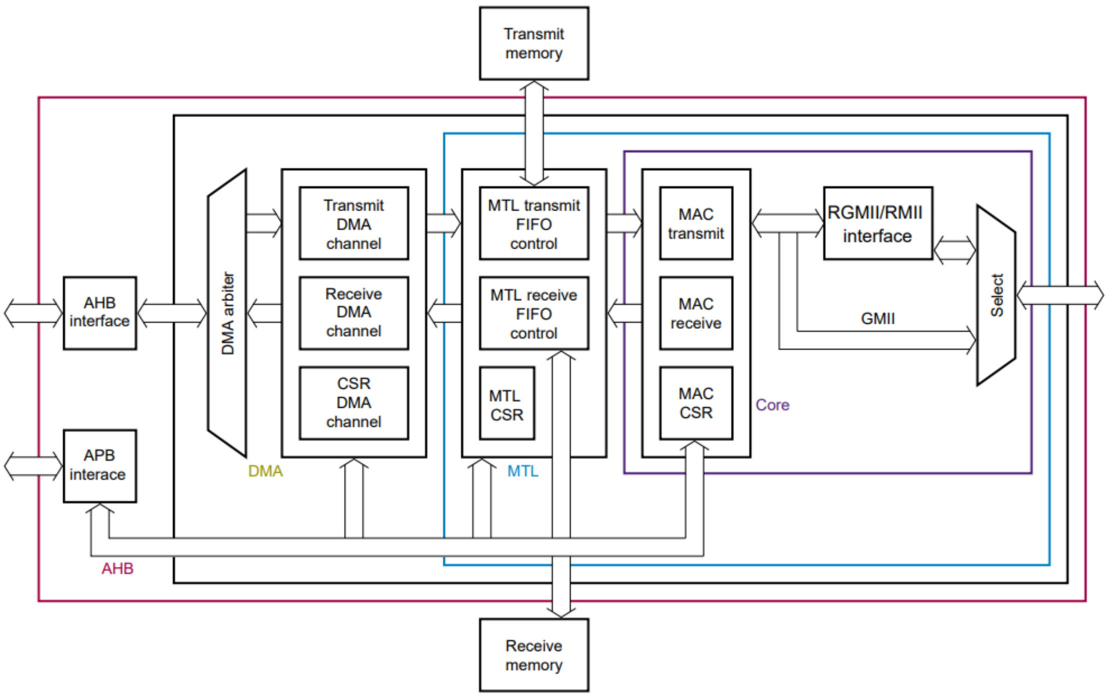


Figure 6.1: EMAC System Level Block Diagram

Below is a detailed overview of the important blocks of the microcontroller or the EMAC architecture that are necessary to understand for Ethernet firmware development, based on the information from the reference manual of the S32K3xx microcontroller.

6.1 EMAC Core Features

6.1.1 MII and RMII Interfaces Support

- These interfaces provide 10/100 Mbps application support in compliance with the IEEE802.3-2015 specifications.
- MII (Media Independent Interface) and RMII (Reduced Media Independent Interface) are standards for connecting MAC to PHY (Physical Layer).

6.1.2 Time-Sensitive Networking and AVB Support

- Supports advanced applications such as time-aware shaping (IEEE802.1bv), time synchronization (IEEE802.1AS-rev), and frame preemption (IEEE802.1Qbu) for time-sensitive networking.
- Also supports media clock recovery and generation for Audio Video Bridging (AVB).

6.1.3 AMBA 2.0 Interface Compatibility

- Provides compatibility with AMBA 2.0 for the AHB master and APB3 interface.

6.2 EMAC Additional Features

6.2.1 Automatic CRC and Pad Generation/Stripping

- The MAC can automatically generate and strip CRC (Cyclic Redundancy Check) and padding in packets.

6.2.2 CRC Checking Control

- Provides the option to disable CRC checking.

6.2.3 Packet Gap Control

- Includes a programmable insert packet gap feature.

6.2.4 Source Address Insertion or Replacement

- Allows for the insertion or replacement of the source address in transmitted packets.

6.2.5 VLAN Tagging and Processing

- Supports VLAN (Virtual Local Area Network) insertion, replacement, and deletion in transmitted packets. It includes control for up to two VLAN tags and queue or channel-based VLAN tags.
- Detects IEEE802.1Q VLAN tags in received packets with an option to delete these tags.

6.2.6 Packet Filtering and Address Filtering

- Offers flexible address filtering modes, including up to two additional 48-bit perfect DA (Destination Address) filters with masks for each byte, up to two 48-bit SA (Source Address) comparison checks with masks for each byte, and a 64-bit hash filter for multicast and unicast DA addresses.
- Provides options for promiscuous mode and passing all multi-cast addressed packets.

6.2.7 Layer 3 and Layer 4 Filtering

- Supports filtering based on Layer 3 (IP) and Layer 4 (TCP/UDP) protocols over IPv4 or IPv6.

6.2.8 Safety and Control Features

- Includes programmable safety watchdog timeout limits and the ability to control the pulse per second (PPS) output signal.

6.2.9 MDIO Interface

- Provides an MDIO (Management Data Input/Output) clause 22 and clause 45 interface for the configuration and management of the PHY device.

6.2.10 Network Statistics and Management

- Supports network statistics with MAC management (RMON) counters.
- **IEEE 1588 Support:** Enables IEEE 1588 sub-nanoseconds support for precise time protocol applications.
- **Frame Preemption and MMC Counters:** Supports frame preemption and provides MMC (Management Counters) for tracking various aspects of MAC operation.

6.3 Microcontroller DMA

The DMA (Direct Memory Access) controller within the S32K3xx microcontroller is a sophisticated component designed to handle high-speed data transfers for network communications, particularly Ethernet packets.

6.3.1 DMA Inclusions

Independent Transmit (Tx) and Receive (Rx) Engines:

- The Tx engine manages data transfers from system memory to the Media Transfer Layer (MTL) interface.
- The Rx engine, conversely, facilitates data movement from the MTL interface to the system memory.
- This design is optimised for packet-oriented data transfers, specifically for Ethernet packets.

Control and Status Register (CSR) Space:

- Communication with the host system is conducted through CSR descriptor lists and data buffers.
- This setup enables efficient management and transfer of data packets.

6.3.2 DMA Descriptors

6.3.2.1 Descriptor Lists and Channels

- The DMA supports up to two transmit and two receive descriptor lists, also known as DMA channels.
- The base address for each list is set in the transmit and receive descriptor list address registers.

6.3.2.2 Descriptor Ring Structure

- Descriptors are forward-linked, with the next descriptor always at a fixed offset from the current one, controlled by the DMA_CH0_Control[DSL].
- The system uses the DMA Channel 0 Tx Descriptor Ring Length (DMA_CH0_TxDesc_Ring_Length) and the DMA Channel 0 Rx Descriptor Ring Length (DMA_CH0_RxDesc_Ring_Length) registers to program the number of descriptors.
- After processing the last descriptor in a list, the DMA jumps back to the descriptor in the list address register, creating a circular or ring structure for efficient processing.

6.3.2.3 Efficient Data Movement

- DMA descriptors are designed for efficient data transfer with minimal intervention from the host.
- The DMA controller can be programmed to issue interrupts in various situations, like the completion of packet transmission and reception.

6.3.2.4 Descriptor Memory Allocation

- Descriptor lists reside in the application's physical memory address space.
- Each descriptor can point to a maximum of two buffers in the system memory, allowing for flexible memory management and avoiding the need for contiguous memory allocation.

6.3.3 Data Buffers

- Data buffers are located in the application's physical memory space.
- Each buffer can contain an entire packet or a part of a packet, but cannot exceed the size of a single packet.

6.3.4 Data and Buffer Status

- Buffers are dedicated solely to data storage.
- The status of each buffer is maintained within the descriptor, ensuring proper tracking and management of the data.

6.3.5 Data Chaining and Packet Handling

- The concept of data chaining allows for packets that span multiple data buffers.
- However, a single descriptor cannot handle multiple packets.
- When an End Of Packet (EOP) signal is detected, the DMA controller moves to the data buffer of the next packet, ensuring efficient and sequential processing of network data.

6.3.6 DMA Controller Bus Burst Access

- When the AHB interface is configured for address-aligned beats, both the transmit and receive DMA engines ensure that the first burst transfer initiated by the AHB is less than or equal to the configured Programmable Burst Length (PBL) value.
- Subsequent beats start at an address aligned to this value.
- The DMA can align the address for beats up to size 16 (for PBL > 16) because it does not support more than INCR16.

6.3.7 DMA Application Data Buffer Alignment

- There are no specific restrictions on the start address alignment for transmit and receive data buffers. For example, in systems with 32-bit memory, the start address for buffers can be aligned to any of the four bytes.
- However, DMA always initiates write transfers with an address aligned to the bus width, and dummy data (old data) is present in the invalid byte lanes.
- This alignment is particularly relevant during the beginning or end of an Ethernet packet transfer, and the software driver must discard the dummy bytes based on the start address of the buffer and the size of the packet.

6.3.8 DMA Buffer Size Calculations

- The DMA does not update the size fields in the transmit and receive descriptors but only the status fields (RDES and TDES).
- The driver is responsible for performing the size calculations.
- The transmit DMA transfers the exact number of bytes indicated by the buffer size field of TDES2 to the MAC.
- For the received DMA, the amount of valid data in a buffer is indicated by the buffer size fields in DMA Channel Rx Control (DMA_CH0_Rx_Control) minus the data buffer pointer offset.
- The offset is zero when the data buffer pointer is aligned to the data bus width.
- For a descriptor marked as last, the buffer may not be full, and the driver must compute the amount of valid data in this final buffer.

6.3.9 DMA Arbiter

The arbiter inside the DMA module performs the arbitration between the transmit and receive channel accesses to the AHB master interface. Two types of arbitrations are supported:

- **Round-Robin:** If DMA_Mode[DA] = 0, and both transmit and receive DMAs simultaneously request access, the arbiter allocates the data bus in ratio sets defined in DMA_Mode[PR].
- **Fixed-Priority:** If DMA_Mode[DA] = 1, the receive DMA is prioritised over the transmit DMA for data access by default. If DMA_Mode[TXPR] = 1, the transmit DMA is prioritised over the receive DMA as defined in the settings.

6.4 MTL Block

The Media Transfer Layer (MTL) in the S32K3xx microcontroller series plays a crucial role in managing data flow between the system memory and the MAC (Media Access Control) block. Here is a detailed account of the MTL, specifically focusing on the transmit path, transmit control word, transmit operation, and the initialization flow:

6.4.1 MTL Overview

- **Function:** Provides a FIFO (First-In-First-Out) memory interface to buffer and regulate packets between system memory and MAC.
- **Data Transfer:** Enables data transfer between the system clock and MAC clock domains.
- **Data Paths:** Has two distinct paths - transmit and receive paths.
- **Communication:** Interacts with the host through ATI (Application Transmit Interface) on the transmit path and ARI (Application Receive Interface) on the receive path.

6.4.2 Transmit Path

- **Internal DMA Handling:** Manages all transactions for the transmit path through ATI, pushing Ethernet packets read from system memory to the corresponding queue.
- **Packet Transfer:** An Ethernet packet is transferred to MAC when the queue reaches its threshold in Threshold mode or if the complete packet is in the queue in Store-and-Forward mode. The End Of Packet (EOP) status is then transferred back to the internal DMA from MAC.

6.4.3 Transmit Control Word

The transmit control word contains essential control information for packet transmission, provided through the ATI interface. It includes:

- Packet length (applicable if DCB is enabled with WFQ scheduling algorithm).
- CRC pad control.
- Source address insertion control.
- VLAN insertion and replacement control, along with VLAN tags for outer and inner VLANs.
- TCP/IP checksum insertion control.
- One-step timestamping control correction.
- Transmit timestamp enable.

6.4.4 Transmit Operation Modes

- **Threshold Mode:** The default mode where data is forwarded to MAC as soon as the number of bytes in the queue crosses a configured threshold level or when the end of a packet is written before the threshold is reached.
- **Store-and-Forward Mode:** In this mode, MTL pops the packet out to MAC only under certain conditions like when a complete packet is stored in the queue, the transmit FIFO is almost full, or the ATI watermark becomes low. This mode allows packet transmission even if the packet length is larger than the transmit queue size.

6.4.5 Initialization Flow

- **Post-Reset State:** MTL is ready to manage data flow between DMA and MAC after a reset.
- **Single-Transmit Queue Configuration:** There are no specific initialization requirements for enabling MTL.
- **Multiple-Transmit Queue Configuration:** Requires initialization of the queue size for each of the queues by programming MTL_TxQ0_Operation_Mode[TQS] corresponding to a transmit queue. The MAC block and internal DMA controllers must also be initialized, with DMA controllers enabled through their respective Control and Status Registers (CSRs).

6.4.6 MTL Receive Path

- **Packet Reception:** The MAC (Media Access Control) block sends packets to the MTL receive module and pushes them into the receive queue.
- **Queue Status Indication:** MTL indicates the status or fill level of the queue to the application or DMA (Direct Memory Access) under two scenarios:
- When the queue's fill level crosses the configured receive threshold, as set in MTL_RxQ0_Operation_Mode[RTC].
- When the MTL receive module receives a complete packet in Store-and-Forward mode.
- **Queue Fill Level:** MTL also communicates the fill level of the queue. This is essential for the DMA to initiate preconfigured burst transfers to the AHB (Advanced High-performance Bus) interface.

6.4.7 MTL Receive Operation

The detailed operation of the MTL in the receive path is as follows:

- **Receiving Packets:** The MAC sends data packets to the MTL receive module, which are then queued for processing.
- **Indicating Queue Status:** MTL monitors and indicates the queue status (fill level) based on two key operational modes:

- **Threshold Mode:** In this mode, MTL signals the application or DMA when the amount of data in the queue reaches a predefined threshold level set in MTL_RxQ0_Operation_Mode[RTC].
- **Store-and-Forward Mode:** Here, MTL waits until it receives a complete packet before indicating the queue status. This mode ensures that only complete packets are forwarded, improving data integrity.
- **Queue Fill Level for DMA:** The fill level of the queue is crucial for managing DMA operations. MTL's indication of the queue fill level allows the DMA to effectively plan and execute burst transfers to the AHB interface, optimizing data transfer and system performance.

6.4.8 Threshold Mode

In the default Threshold mode, MTL reads data and signals its availability to the application or DMA under two conditions:

- When the data bytes in the receive queue reach the amount set as the threshold in MTL_RxQ0_Operation_Mode[RTC] and MTL_RxQ1_Operation_Mode[RTC].
- When a full packet of data is received into the queue.

6.4.9 Store-and-Forward Mode

- **Functionality:** In Store-and-Forward mode (activated when MTL_RxQ0_Operation_Mode[RSF] = 1), the initial locations in the receive queue are reserved for status words before the start of packet (SOP) is written.
- **Packet Handling:** A packet is read out only after it is completely written into the receive queue. This mode ensures that only complete and valid packets are processed and forwarded, enhancing data integrity.
- **Error Packet Handling:** All error packets are dropped if configured through MTL_RxQ0_Operation_Mode[FEP], ensuring that only valid packets are read and forwarded to the application.

6.5 MAC Block

The MAC (Media Access Control) in the S32K3xx microcontroller series supports both the MII (Media Independent Interface) and RMII (Reduced Media Independent Interface) PHY interfaces. It consists of three main components: MTI (MAC Transmit Interface), MRI (MAC Receive Interface), and MCI (MAC Control Interface). Here is a detailed overview of both MAC transmission and reception processes:

6.5.1 MAC Transmission Process

- **Initiation of Transmission:** The transmission process begins when the Media Transfer Layer (MTL) pushes in data with the Start Of Packet (SOP) signal asserted.

- **Data Acceptance and Transmission:** After the SOP signal is detected, the MAC accepts the data and begins transmitting it to either the RMII or MII interface.
- **Completion of Transmission:** Upon receiving the End Of Packet (EOP) signal, the MAC performs one of the following steps,
    - Completes the normal packet transmission and sends the transmission status back to the MTL.
    - In cases of a normal collision (specifically in Half-Duplex mode), the MAC sends retry requests during transmission until either the packet is successfully transmitted or the maximum number of retry requests is exhausted.

6.5.2 MAC Reception Process

- **Detection of Frame Data:** The MAC initiates the receive operation by detecting a state-of-frame data on the RMII or MII interface.
- **Preamble and SFD Stripping:** It then strips the preamble and Start Frame Delimiter (SFD) before processing the Ethernet packet.
- **Address Filtering Management (AFM):** The MAC's AFM checks the header fields (Source Address - SA and Destination Address - DA) of the incoming packet for filtering. It also verifies the Cyclic Redundancy Check (CRC) contained in the packet's Frame Check Sequence (FCS) field.
- **Storage of Received Packet:** The MAC stores the received packet in a shallow buffer until address filtering is completed.
- **Dropping Packets Failing Address Filter:** If a packet fails the address filter, the AFM drops it.

6.6 Interrupts

The Ethernet Media Access Controller (EMAC) in the S32K3xx microcontroller series incorporates a sophisticated interrupt system designed to efficiently handle various network events and reduce CPU load. Here is a detailed account of the interrupt system:

6.6.1 Interrupt Coalescing

- **Functionality:** EMAC supports interrupt coalescing, which reduces the number of interrupts generated by the module, thereby lowering CPU load.
- **Interrupt Status Tracking:** The MAC Interrupt Status (MAC_Interrupt_Status) register captures various interrupt events. The generation of an interrupt is contingent upon the corresponding interrupt enable field being set to 1, and is based on the event status in the Status registers.
- **Interrupt Mode (INTM) Field:** This field determines whether the interrupt signal is a level signal or a pulse signal.

6.6.2 Interrupt Requests

- **Common Interrupts:** The MAC Interrupt Enable (MAC_Interrupt_Enable) register and the MAC Interrupt Status (MAC_Interrupt_Status) register are used for common interrupts.
- **MMC Receive Interrupts:** These are controlled by the MMC Receive Interrupt Mask (MMC_Rx_Interrupt_Mask) and the MMC Receive Interrupt (MMC_Rx_Interrupt).
- **MMC FPE Receive Interrupts:** These are governed by the MMC FPE Receive Interrupt Mask (MMC_FPE_Rx_Interrupt_Mask) and the MMC Receive Packet Assembly Error Counter Interrupt Status (MMC_FPE_Rx_Interrupt).
- **MTL Debug and EST Interrupts:** Managed by the MTL Debug Control (MTL_DBG_CTL), MTL Debug Status (MTL_DBG_STS), MTL EST Interrupt Enable (MTL_EST_Intr_Enable), and MTL EST Status (MTL_EST_Status).
- **MTL Rx Parser Interrupts:** Controlled through the MTL Rx Parser Interrupt Control Status (MTL_RXP_Interrupt_Control_Status).

6.6.3 DMA Channel Interrupts

- **Per Channel Interrupts:** Each receive channel has a dedicated interrupt (sbd_perch_rx_intr_o), and the number of these interrupts corresponds to the number of transmit/receive queues (max_dma_ch), which is 2.
- **Common Interrupts:** The sbd_intr_o common interrupt is a level signal that activates when a corresponding interrupt event source is present in the DMA Interrupt Status (DMA_Interrupt_Status) register. This register contains event source fields corresponding to each DMA channel, the MAC transaction layer, and MAC blocks.

6.6.4 Transfer Complete Interrupt Behaviour

- **Interrupt Mode (INTM):** When INTM is set to 1, the signals indicate the values of the corresponding DMA_CH0_Status[RI] and DMA_CH0_Status[TI] fields when these fields are set to 1. These signals are level signals that are cleared by writing 1 to these fields. They do not assert when DMA_CH0_Status[RI] or DMA_CH0_Status[TI] is 0.

This interrupt system, with its various components and functionalities, is integral to the efficient and effective management of network events and data processing within the EMAC of the S32K3xx microcontroller series. It ensures timely and appropriate responses to network conditions, thereby optimizing the controller's performance and reliability.

6.7 External Module Signals

6.7.1 MII_RMII_TXCLK (Clock)

- **Type: Input (I)**

- **Description**
  - In the MII (Media Independent Interface) configuration, this transmission clock is provided by the external PHY (Physical Layer).
  - The clock operates at a frequency of 25 MHz in 100 Mbps mode and at 2.5 MHz in 10 Mbps mode.
  - All transmission signals generated by the MAC (Media Access Control) are synchronized with this clock.
  - This clock signal is essential for all PHY interfaces, ensuring proper timing and synchronization for data transmission.

6.7.2 MII_RX_CLK (Clock)

- **Type: Input (I)**
- **Description**
  - This clock is for the MII and RMII (Reduced Media Independent Interface) interfaces, provided by the external PHY. The clock operates at 25 MHz in 100 Mbps mode and at 2.5 MHz in 10 Mbps mode. All MII receive signals that MAC receives are synchronous to MII_RX_CLK.

6.7.3 EMAC_PPS[3:0] (Signals)

- **Type: Input/Output (I/O)**
- **Description**
  - This group of signals is used as pulse per second in Output mode and as media clock generation trigger in Input mode.
  - They trigger input to the Device Under Test (DUT) to capture presentation time.
  - The signals can be defined as pulse or level signals based on the presentation control value of MAC PPS Control (MAC_PPS_Control).

6.7.4 MII_RMII_TX_EN (Signal)

- **Type: Output (O)**
- **Description**
  - This signal is driven by MAC and performs multiple functions depending on the selected PHY interface.
  - In MII mode, it indicates that valid data is being transmitted to the phy_txd_o bus and is synchronous to MII_RMII_TX_CLK.
  - In RMII mode, it indicates that valid data is being transmitted to the phy_txd_o bus and is synchronous to MII_RMII_RX_CLK.

6.7.5 MII_RMII_TXD[3:0] (Signals)

- **Type: Output (O)**

- **Description**
  - This is a group of transmit data signals driven by MAC.
  - They perform multiple functions depending on the selected PHY interface.
  - In MII mode, bits [3:0] provide the MII transmit data nibble, which is valid only when the signal is high. It is synchronous to MII_RMII_TX_CLK.
  - In RMII mode, bits [1:0] provide the RMII transmit data. The data is valid only when the signal is high. It is synchronous to MII_RMII_TX_CLK.

6.7.6 MII_CRS (Signal)

- **Type: Input (I)**
- **Description**
  - Valid only in MII (Media Independent Interface) mode.
  - The PHY (Physical Layer) drives this signal high when the transmit or receive medium is not idle and low when both mediums are idle.
  - The signal is not synchronous to any clock.

6.7.7 MII_COL (Signal)

- **Type: Input (I)**
- **Description**
  - Valid only in MII mode.
  - The PHY drives this signal high when a collision is detected on the medium.
  - This signal is also not synchronous to any clock.

6.7.8 MII_RMII_RX_DV (Signal)

- **Type: Input (I)**
- **Description**
  - Driven by the PHY.
  - It performs multiple functions depending on the selected PHY interface.
  - In MII mode, it indicates that the data on the MII_RXD bus is valid. It remains high continuously from the first recovered byte or nibble of the packet through the final recovered byte or nibble of the packet. It is synchronous to MII_RX_CLK.
  - In RMII mode, it contains the CRS and data valid information of the receive interface. It is synchronous to MII_RMII_TX_CLK.

6.7.9 MII_RMII_RX_ER (Signal)

- **Type: Input (I)**
- **Description**

    - In MII mode, indicates an error or carrier extension in the received packet of the MII_RXD[3:0] bus. It is synchronous to MII_RX_CLK.
    - Not used in RMII mode.

6.7.10 MII_RMII_RXD[3:0] (Signals)

- **Type: Input (I)**
- **Description**
    - These are a group of data signals received from the PHY.
    - In MII mode, bits [3:0] provide the MII receive data nibble, valid only when the MII_RMII_RX_DV signals are high. They are synchronous to MII_RX_CLK.
    - In RMII mode, bits [1:0] provide the RMII receive data, valid only when the MII_RMII_RX_DV signal is high. They are synchronous to MII_RMII_TX_CLK.

6.7.11 MII_RMII_MDC

- **Type: Output (O)**
- **Description**
    - MAC provides timing reference for MII_RMII_MDIO or MII through this periodic clock.
    - The application clock generates this clock through a clock divider controlled by MAC_MDIO_Address[CR].

6.7.12 MII_RMII_MDIO (Signal)

- **Type: Input/Output (I/O)**
- **Description**
    - MDIO uses this signal to transfer control and data information to PHY.

6.8 PHY Interfaces

The S32K344 microcontroller series supports the use of PHY (Physical Layer) interfaces, providing essential communication capabilities for networking applications. Here's a detailed overview of how the PHY interfaces are utilized in this module:

6.8.1 Supported Modes

6.8.1.1 RMII 10/100 Mbps Interface:

- The Reduced Media Independent Interface (RMII) operates at speeds of 10/100 Mbps.

- It is a more streamlined version of the MII interface, requiring fewer data lines for operation.

6.8.1.2 MII 10/100 Mbps Interface:

- The Media Independent Interface (MII) also operates at 10/100 Mbps speeds.
- It is a standard interface used to connect a Fast Ethernet (i.e., 100 Mbps) MAC block to a PHY chip.

6.8.2 Interface Selection

- **Phy_intf_sel Input Signal:** This signal determines which mode (RMII or MII) is selected. The signal is sampled at reset.
- **Configuration Registers:** The MAC_Configuration[PS] and MAC_Configuration[FES] registers are used to select the operating speed of the chosen mode.

6.8.3 RMII Reference Clock

- The RMII reference clock, which operates at 50 MHz, can be fed to the IP (Intellectual Property) either from an external source or internally from a Phase-Locked Loop (PLL) on the System on Chip (SoC).

6.8.4 PHY Register Access

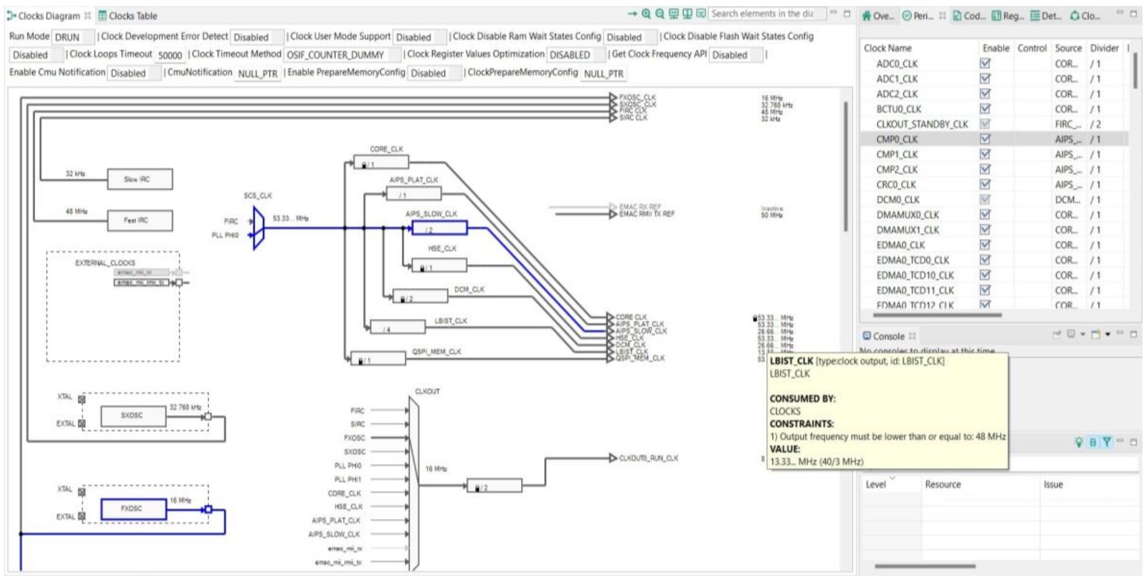
The module provides access to the PHY registers through the Station Management Interface (SMA), a two-wire interface consisting of:

- **MDC (Management Data Clock):** A clock line for the MDIO interface.
- **MDIO (Management Data Input/Output):** A data line for bidirectional data transfer with the PHY.
- **Operating Frequency of MDC:** According to the IEEE 802.3 specification, the maximum operating frequency of MDC is 2.5 MHz. The system clock derives this frequency using a divider.
- **MDC Clock Frequency Configuration:** The MAC_MDIO_Address[CR] register programs the generation of different MDC clock frequencies.

7 Microcontroller Clocks and Port Configuration and Pin Assignments

This section outlines the process of various clock configurations, port configuration settings and pin assignments applied to the S32K344 microcontroller as per requirements of the Ethernet MAC (Media Access Controller) to enable ethernet communication.

In Ethernet applications, a precise clock configuration is essential for managing data transmission rates and interfacing with PHY devices through RMII or MII interfaces. The figures detail how the PLL (Phase-Locked Loop) is configured to generate the required clock frequencies for the Ethernet MAC (Media Access Controller) to operate correctly under IEEE 802.3 standards.



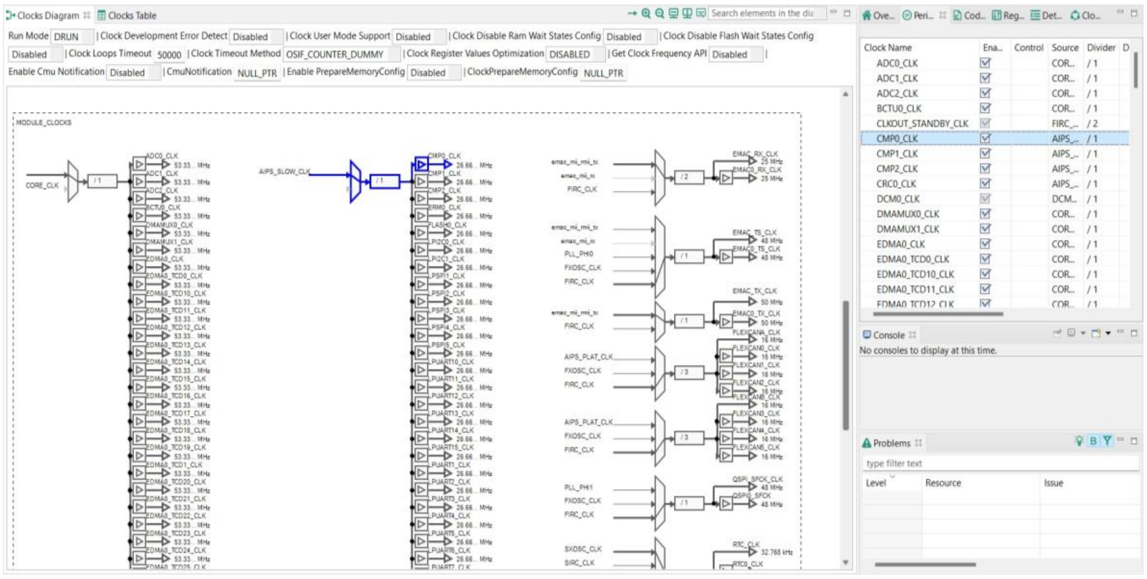
Run Mode DRUN | Clock Development Error Detect Disabled | Clock User Mode Support Disabled | Clock Disable Ram Wait States Config Disabled | Clock Disable Flash Wait States Config Disabled | Clock Loops Timeout 50000 | Clock Timeout Method OSIF_COUNTER_DUMMY | Clock Register Values Optimization DISABLED | Get Clock Frequency API Disabled |
Enable Cmu Notification Disabled | CmuNotification NULL_PTR | Enable PrepareMemoryConfig Disabled | ClockPrepareMemoryConfig NULL_PTR

Clock Name	Enable	Control	Source	Divider
ADC0_CLK	✓		COR...	/ 1
ADC1_CLK	✓		COR...	/ 1
ADC2_CLK	✓		COR...	/ 1
BCTU0_CLK	✓		COR...	/ 1
CLKOUT_STANDBY_CLK	✓		FIRC_...	/ 2
CMP0_CLK	✓		AIPS_...	/ 1
CMP1_CLK	✓		AIPS_...	/ 1
CMP2_CLK	✓		AIPS_...	/ 1
CRC0_CLK	✓		AIPS_...	/ 1
DCM0_CLK	✓		DCM...	/ 1
DMAMUX0_CLK	✓		COR...	/ 1
DMAMUX1_CLK	✓		COR...	/ 1
EDMA0_CLK	✓		COR...	/ 1
EDMA0_TCD0_CLK	✓		COR...	/ 1
EDMA0_TCD10_CLK	✓		COR...	/ 1
EDMA0_TCD11_CLK	✓		COR...	/ 1
EDMA0_TCD12_CLK	✓		COR...	/ 1

LBIST_CLK [type:clock output, id: LBIST_CLK]
LBIST_CLK

CONSUMED BY:
CLOCKS
CONSTRAINTS:
1) Output frequency must be lower than or equal to: 48 MHz
VALUE:
13.33... MHz (40/3 MHz)

Figure 7.1: EMAC Clock Configuration 1

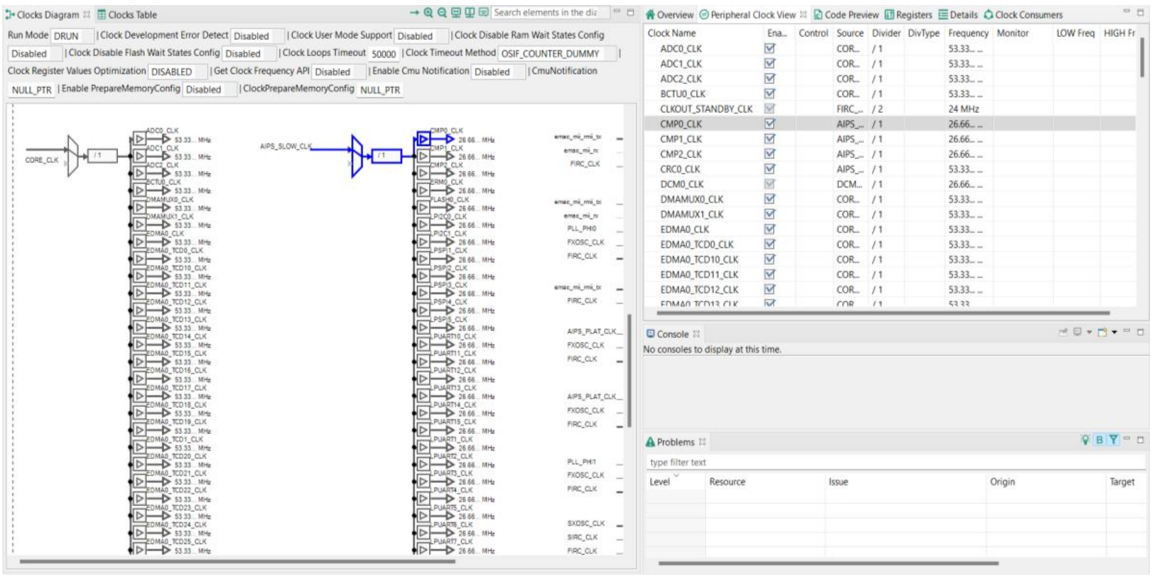


Run Mode DRUN | Clock Development Error Detect Disabled | Clock User Mode Support Disabled | Clock Disable Ram Wait States Config Disabled | Clock Disable Flash Wait States Config Disabled | Clock Loops Timeout 50000 | Clock Timeout Method OSIF_COUNTER_DUMMY | Clock Register Values Optimization DISABLED | Get Clock Frequency API Disabled |
Enable Cmu Notification Disabled | CmuNotification NULL_PTR | Enable PrepareMemoryConfig Disabled | ClockPrepareMemoryConfig NULL_PTR

Clock Name	Ena...	Control	Source	Divider
ADC0_CLK	✓		COR...	/ 1
ADC1_CLK	✓		COR...	/ 1
ADC2_CLK	✓		COR...	/ 1
BCTU0_CLK	✓		COR...	/ 1
CLKOUT_STANDBY_CLK	✓		FIRC_...	/ 2
CMP0_CLK	✓		AIPS_...	/ 1
CMP1_CLK	✓		AIPS_...	/ 1
CMP2_CLK	✓		AIPS_...	/ 1
CRC0_CLK	✓		AIPS_...	/ 1
DCM0_CLK	✓		DCM...	/ 1
DMAMUX0_CLK	✓		COR...	/ 1
DMAMUX1_CLK	✓		COR...	/ 1
EDMA0_CLK	✓		COR...	/ 1
EDMA0_TCD0_CLK	✓		COR...	/ 1
EDMA0_TCD10_CLK	✓		COR...	/ 1
EDMA0_TCD11_CLK	✓		COR...	/ 1
EDMA0_TCD12_CLK	✓		COR...	/ 1

Console
No consoles to display at this time.

Figure 7.2: EMAC Clock Configuration 2

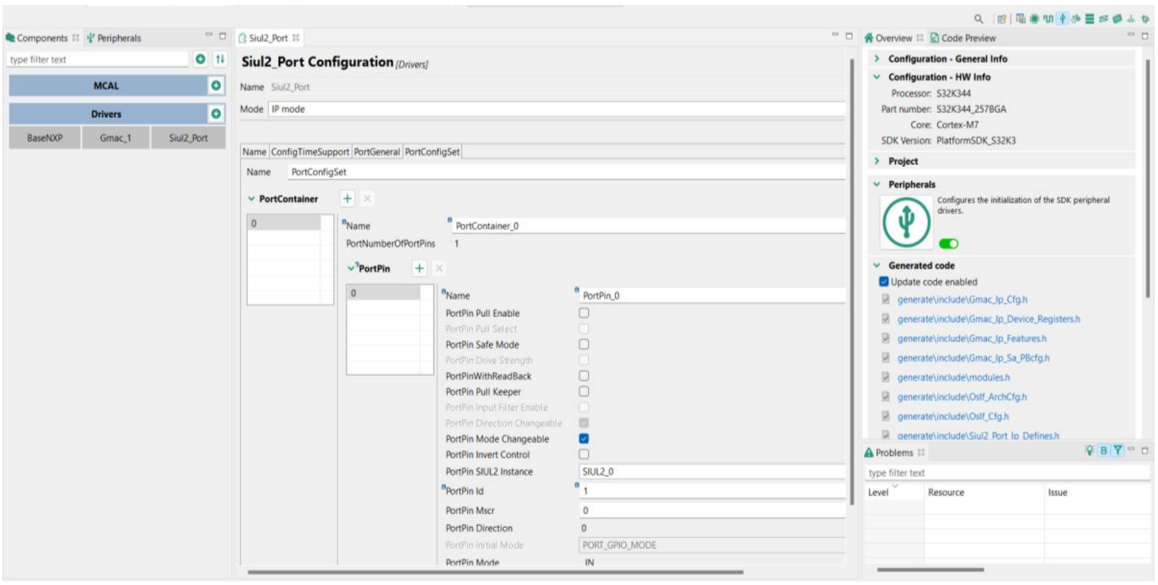

Run Mode DRUN | Clock Development Error Detect Disabled | Clock User Mode Support Disabled | Clock Disable Ram Wait States Config Disabled | Clock Disable Flash Wait States Config Disabled | Clock Loops Timeout 50000 | Clock Timeout Method OSIF_COUNTER_DUMMY | Clock Register Values Optimization DISABLED | Get Clock Frequency API Disabled | Enable Cmu Notification Disabled | CmuNotification NULL_PTR | Enable PrepareMemoryConfig Disabled | ClockPrepareMemoryConfig NULL_PTR

Clock Name	Ena...	Control	Source	Divider	DivType	Frequency	Monitor	LOW Freq	HIGH Fr
ADC0_CLK	✓		COR...	/ 1		53.33... ...			
ADC1_CLK	✓		COR...	/ 1		53.33... ...			
ADC2_CLK	✓		COR...	/ 1		53.33... ...			
BCTU0_CLK	✓		COR...	/ 1		53.33... ...			
CLKOUT_STANDBY_CLK	✓		FIRC_...	/ 2		24 MHz			
CMP0_CLK	✓		AIPS_...	/ 1		26.66... ...			
CMP1_CLK	✓		AIPS_...	/ 1		26.66... ...			
CMP2_CLK	✓		AIPS_...	/ 1		26.66... ...			
CRC0_CLK	✓		AIPS_...	/ 1		53.33... ...			
DCM0_CLK	✓		DCM...	/ 1		26.66... ...			
DMAMUX0_CLK	✓		COR...	/ 1		53.33... ...			
DMAMUX1_CLK	✓		COR...	/ 1		53.33... ...			
EDMA0_CLK	✓		COR...	/ 1		53.33... ...			
EDMA0_TCD0_CLK	✓		COR...	/ 1		53.33... ...			
EDMA0_TCD10_CLK	✓		COR...	/ 1		53.33... ...			
EDMA0_TCD11_CLK	✓		COR...	/ 1		53.33... ...			
EDMA0_TCD12_CLK	✓		COR...	/ 1		53.33... ...			

No consoles to display at this time.

Figure 7.3: EMAC Clock Configuration 3

Various I/O ports on the S32K344 microcontroller were also configured to allow interfacing with the Ethernet PHY and enable Ethernet functionality.


Siul2_Port Configuration [Drivers]

Name Siul2_Port
Mode IP mode

Name PortConfigSet

PortContainer
Name PortContainer_0
PortNumberOfPortPins 1

PortPin
Name PortPin_0
PortPin Pull Enable ☐
PortPin Pull Select ☐
PortPin Safe Mode ☐
PortPin Drive Strength ☐
PortPinWithReadBack ☐
PortPin Pull Keeper ☐
PortPin Input Filter Enable ☐
PortPin Direction Changeable ✓
PortPin Mode Changeable ✓
PortPin Invert Control ☐
PortPin SIUL2 Instance SIUL2_0
PortPin Id 1
PortPin Mscr 0
PortPin Direction 0
PortPin Initial Mode PORT_GPIO_MODE
PortPin Mode IN

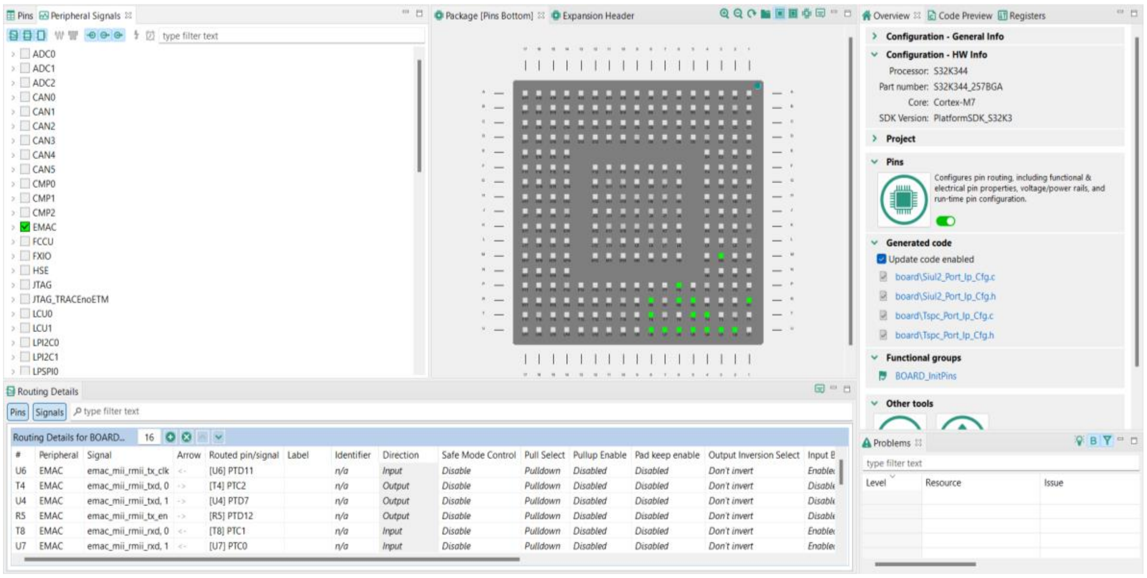
Configuration - HW Info
Processor: S32K344
Part number: S32K344_257BGA
Core: Cortex-M7
SDK Version: PlatformSDK_S32K3

Peripherals
Configures the initialization of the SDK peripheral drivers.

Generated code
Update code enabled
generate\include\Gmac_Ip_Cfg.h
generate\include\Gmac_Ip_Device_Registers.h
generate\include\Gmac_Ip_Features.h
generate\include\Gmac_Ip_Sa_PBcfg.h
generate\include\modules.h
generate\include\OsIf_ArchCfg.h
generate\include\OsIf_Cfg.h
generate\include\Siul2_Port_Ip_Defines.h

Figure 7.4: Port Configuration

Configurations such as pin assignments required to utilize Ethernet capabilities required allocation of pins for RMII and MII data lines, for TX (transmit) and RX (receive) lines, management data clocks (MDC), and management data input/output (MDIO) lines for PHY configuration and status monitoring. These pin assignments are critical for ensuring that the microcontroller can communicate effectively with the Ethernet PHY, manage data transmission and reception, and maintain synchronization with network protocols.



Processor: S32K344
Part number: S32K344_257BGA
Core: Cortex-M7
SDK Version: PlatformSDK_S32K3

Configures pin routing, including functional & electrical pin properties, voltage/power rails, and run-time pin configuration.

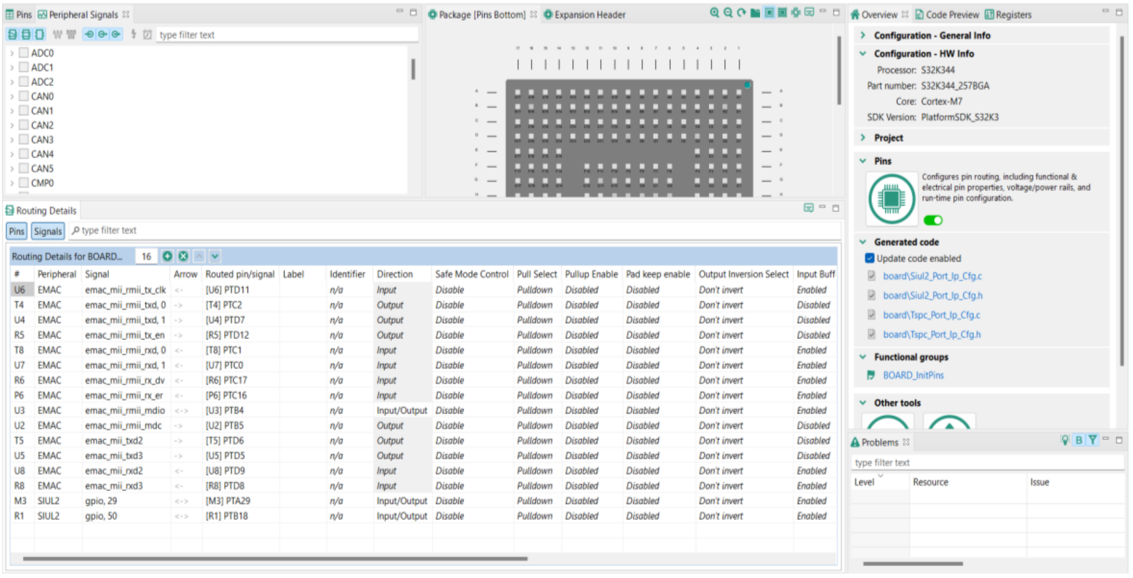
Generated code
Update code enabled
board\Siul2_Port_Ip_Cfg.c
board\Siul2_Port_Ip_Cfg.h
board\Tspc_Port_Ip_Cfg.c
board\Tspc_Port_Ip_Cfg.h

Functional groups
BOARD_InitPins

Routing Details for BOARD... 16

#	Peripheral	Signal	Arrow	Routed pin/signal	Label	Identifier	Direction	Safe Mode Control	Pull Select	Pullup Enable	Pad keep enable	Output Inversion Select	Input B
U6	EMAC	emac_mii_rmii_tx_clk	<-	[U6] PTD11		n/a	Input	Disable	Pulldown	Disabled	Disabled	Don't invert	Enablec
T4	EMAC	emac_mii_rmii_txd, 0	->	[T4] PTC2		n/a	Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Disable
U4	EMAC	emac_mii_rmii_txd, 1	->	[U4] PTD7		n/a	Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Disable
R5	EMAC	emac_mii_rmii_tx_en	->	[R5] PTD12		n/a	Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Disable
T8	EMAC	emac_mii_rmii_rxd, 0	<-	[T8] PTC1		n/a	Input	Disable	Pulldown	Disabled	Disabled	Don't invert	Enablec
U7	EMAC	emac_mii_rmii_rxd, 1	<-	[U7] PTC0		n/a	Input	Disable	Pulldown	Disabled	Disabled	Don't invert	Enablec

Figure 7.5: Pin Assignments 1



Routing Details for BOARD... 16

#	Peripheral	Signal	Arrow	Routed pin/signal	Label	Identifier	Direction	Safe Mode Control	Pull Select	Pullup Enable	Pad keep enable	Output Inversion Select	Input Buff
U6	EMAC	emac_mii_rmii_tx_clk	<-	[U6] PTD11		n/a	Input	Disable	Pulldown	Disabled	Disabled	Don't invert	Enabled
T4	EMAC	emac_mii_rmii_txd, 0	->	[T4] PTC2		n/a	Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Disabled
U4	EMAC	emac_mii_rmii_txd, 1	->	[U4] PTD7		n/a	Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Disabled
R5	EMAC	emac_mii_rmii_tx_en	->	[R5] PTD12		n/a	Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Disabled
T8	EMAC	emac_mii_rmii_rxd, 0	<-	[T8] PTC1		n/a	Input	Disable	Pulldown	Disabled	Disabled	Don't invert	Enabled
U7	EMAC	emac_mii_rmii_rxd, 1	<-	[U7] PTC0		n/a	Input	Disable	Pulldown	Disabled	Disabled	Don't invert	Enabled
R6	EMAC	emac_mii_rmii_rx_dv	<-	[R6] PTC17		n/a	Input	Disable	Pulldown	Disabled	Disabled	Don't invert	Enabled
P6	EMAC	emac_mii_rmii_rx_er	<-	[P6] PTC16		n/a	Input	Disable	Pulldown	Disabled	Disabled	Don't invert	Enabled
U3	EMAC	emac_mii_rmii_mdio	<->	[U3] PTB4		n/a	Input/Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Enabled
U2	EMAC	emac_mii_rmii_mdc	->	[U2] PTB5		n/a	Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Disabled
T5	EMAC	emac_mii_txd2	->	[T5] PTD6		n/a	Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Disabled
U5	EMAC	emac_mii_txd3	->	[U5] PTD5		n/a	Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Disabled
U8	EMAC	emac_mii_rxd2	<-	[U8] PTD9		n/a	Input	Disable	Pulldown	Disabled	Disabled	Don't invert	Enabled
R8	EMAC	emac_mii_rxd3	<-	[R8] PTD8		n/a	Input	Disable	Pulldown	Disabled	Disabled	Don't invert	Enabled
M3	SIUL2	gpio, 29	<->	[M3] PTA29		n/a	Input/Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Enabled
R1	SIUL2	gpio, 50	<->	[R1] PTB18		n/a	Input/Output	Disable	Pulldown	Disabled	Disabled	Don't invert	Enabled

Figure 7.6: Pin Assignments 2

This overall process is essential for configuring the S32K344 microcontroller for Ethernet communication, with precise setup of clocks, ports, and pins for multiple peripherals.

8 Firmware Development

This section provides the detailed implementation of the firmware for the whole architecture as described above including initializing of each important block as well as important functions for receive and transmit operations.

8.1 Initializing ENET

This function is responsible for initializing the Ethernet controller. It performs several critical setup tasks, including disabling the MPU (Memory Protection Unit) to allow the Ethernet module access to the memory, setting up the transmit buffer protection semaphore, and configuring the Ethernet controller with the appropriate settings for operation (such as MII/RMII mode, buffer sizes, and MAC addresses).

```
Std_ReturnType Enet_Init()
{
    Std_ReturnType xRet = E_OK;
    Enet_DMA_Init();
    Enet_MAC_Init();
    Enet_MTL_Init();
    return xRet;
}
```

8.2 Initializing DMA

This function initializes the DMA (Direct Memory Access) controller for the Ethernet interface, which is used for efficient data transfer between memory and the Ethernet MAC without involving the CPU for data movement, reducing the CPU load. It first initiates a software reset of the DMA by setting a bit in the **DMA_Mode** register, and then waits for the reset process to complete. Then, it configures the DMA System Bus mode for optimal data transfers. This involves enabling address-aligned beats and mixed burst mode for handling various burst lengths during data transfers. It also sets up individual DMA channels for transmit and receive operations, including burst lengths and the number of descriptors. And finally, it utilizes the **EnetBuffersInit** function to initialize and link the transmit and receive buffer descriptors, also setting up tail pointers for the DMA to know the end of the descriptor lists.

```
static void Enet_DMA_Init(void)
{
/*providing a software reset by set the 0 bit of DMA Mode register*/
    IP_EMAC-> DMA_Mode |= 0x1U;
/* initializing dma sysbus mode */
/* When the AHB interface is configured for address-aligned beats .The DMA can
only align the address for beats up to size 16 for the AHB interface because
it does not support

    more than INCR16.
    pg no 2936*/

#if (IP_EMAC-> DMA_Mode & 0x1U == 0x0U)       /* wait for reset process*/
     IP_EMAC-> DMA_SysBus_Mode = 0x1000U  /* enabling address aligned beats */
      |0x4000U;     /* enabling mixed burst mode for burst length 16 or more*/
/* Check next 5 lines again for alignment  */
    IP_EMAC-> DMA_CH0_Control &= 0xFFFEFFFFu        /* disabling PBLx8 for set
burst length more than 16 */
```

```
                              | 0x100000u;           /*Set the DSL value
depending on the 128-bit to skip between two unchained descriptors
*/
    IP_EMAC-> DMA_CH0_Rx_Control |= 0x100000u;          /* set receive burst
length 16 */
    IP_EMAC-> DMA_CH0_Tx_Control |= 0x100000u;          /* set transfer burst
length 16 */
    IP_EMAC-> DMA_CH0_RxDesc_Ring_Length = 0x4U;        /* 4 Rx descriptors */
    IP_EMAC-> DMA_CH0_TxDesc_Ring_Length = 0x4U;        /* 4 Tx descriptors */

    EnetBuffersInit();

/* set TX/RX pointer to the first descriptor*/

uint32_t* Tail_Ptr_TX = (&EnetTxDescriptors_Normal[4] + sizeof(int));       /*
Tail pointer of TX , last descriptor + 32*/
uint32_t* Tail_Ptr_RX = (&EnetRxDescriptors_Normal[4] + sizeof(int));       /*
Tail pointer of RX , last descriptor + 32*/

/*Initialize the receive and transmit descriptor list address with the base
address of the transmit and receive descriptor. Also, program the transmit and
receive tail pointer registers that indicates DMA about the available
descriptors */

IP_EMAC-> DMA_CH0_TxDesc_Tail_Pointer = (uint32_t) (Tail_Ptr_TX &
0xFFFFFFFCu);
IP_EMAC-> DMA_CH0_RxDesc_Tail_Pointer = (uint32_t) (Tail_Ptr_RX &
0xFFFFFFFCu);

/* Programing the DMA_CH(0)_Interrupt_Enable as normal interrupt summary */
IP_EMAC-> DMA_CH0_Interrupt_Enable |= 0x00008000u;

/* to start the receive and transmit DMAs */
IP_EMAC-> DMA_CH0_Rx_Control |= 0x1u;
IP_EMAC-> DMA_CH0_Tx_Control |= 0x1u;

uint32 bit14 = IP_EMAC-> DMA_SysBus_Mode & 0x4000u;   /* checking of mixed
burst mode enable if so disable fixed burst mode */
#if ( bit14 == 0x4000u )
    IP_EMAC-> DMA_SysBus_Mode &= 0xFFFFFFFEu;  /* disabling fixed burst mode*/
#endif

#endif
}
```

8.3 Initializing MTL registers

This function initializes the MTL (MAC Transmission Logic) registers, which is important as the MTL is responsible for managing the queues for packet transmission and reception. The MTL configuration ensures efficient data handling and flow control at the MAC layer. The function sets the operation mode for the MTL, affecting how it handles transmit and receive queues. It then configures the mapping between receive queues and DMA channels, allowing for efficient packet routing based on destination addresses or filter settings. Finally, the function sets up the transmit and receive queues, including enabling store-and-forward modes (which ensure that a packet is stored entirely before forwarding) and defining queue sizes. This final part of the setup ensures that the queues can handle the intended volume of data traffic efficiently.

```
static void Enet_MTL_Init(void)
{
    /* Initializing MTL operation for multiple transit and receive queues */
    IP_EMAC-> MTL_Operation_Mode &= 0xFFFFFF99u;

    /* If this field is 1, it indicates that the packets received in queue 1
are routed to a particular DMA channel as decided in the MAC receiver based on
the DMA channel number programmed in the L3-L4 filter registers, or the
Ethernet DA address. */

    IP_EMAC-> MTL_RxQ_DMA_Map0 |= 0x00001010u;          /* Queue 0 & Queue 1
Enabled for DA-based DMA Channel Selection*/
    IP_EMAC-> MTL_TxQ0_Operation_Mode  |= 0x0000000Au    /* Transmit Queue
Enable  & Transfer store and forward mode is selected*/
                                       |= 0x000F0000u;      /* allocate queue size
of 4096 (4K) bytes , each queue according to datasheet is of 256 bytes of
block so 16 queues*/

    IP_EMAC-> MTL_RxQ0_Operation_Mode  |=0x00000020u     /*Enable Receive
Queue Store and Forward */
                                        |= 0x00024000u       /*6- Full minus 4 KB,
that is full 4KB Threshold for Deactivating Flow Control (in half-duplex and
full-duplex modes) see data sheet*/
                                        |= 0x00000010u       /*it indicates that
all packets except the runt error packets are forwarded to the application or
DMA. */
                                        |= 0x00F00000u;      /*Receive Queue Size
4K bytes */
}
```

8.4 Initializing MAC

The MAC initialization function is responsible for initializing the MAC (Media Access Control) layer of the Ethernet interface. It configures the MAC address, packet filtering, flow control, and interrupts settings. The device's MAC address is set up by enabling the MAC address in the **MAC_Address0_High** register and setting the upper 16 bits of the MAC address in this register. The **MAC_Address0_Low** register contains the lower 32 bits of the MAC address. Afterwards, the MAC packet filter settings are configured to disable certain types of packet filtering, such as Hash or Perfect Filter and Source or Destination Address Filter. This is done by modifying the **MAC_Packet_Filter** register. The function also disables transmit flow control by adjusting the **MAC_Q0_Tx_Flow_Ctrl** register. This setting affects how the MAC deals with outgoing data flow, preventing it from overwhelming network peers. Finally, various MAC-related interrupts (like PHY status changes, timestamp triggers, and frame transmission/reception status) are enabled by setting bits in the **MAC_Interrupt_Enable** register.

```
static void Enet_MAC_Init(void)
{
   IP_EMAC -> MAC_Address0_High |= 0x80000000u       /* Enables MAC address */
                                |= 0x00010000u;    /*DMA Channel Select, 1 channel */

/* 15-0 ADDRHI. Contains the upper 16 bits [47:32] of the first 6-byte
Destination MAC address */

    IP_EMAC -> MAC_Address0_Low         /* Contains the lower 32 bits of the
first 6-byte MAC address */
    IP_EMAC -> MAC_Packet_Filter &= 0xFFFFFBFFu     /* disable Hash Or Perfect
Filter and Source or destination Address Filter Enable */
                                 &= 0x7FFFFE00u;        /* Disable Unicast,
multicast, broadcast, and control frames filter settings */
    IP_EMAC -> MAC_Q0_Tx_Flow_Ctrl &=0xFFFFFFFDu;   /*Transmit Flow Control
disable */

/* Enable all intrupts together :PHY,Timestamp,Transmit Status,Receive
Status,Frame Preemption,MDIO Interrupt */
    IP_EMAC -> MAC_Interrupt_Enable |= 0x00064008u;
}
```

8.5 Setting ENET MAC Address

This important function sets the MAC (Media Access Control) address for the Ethernet controller. This address is crucial for the protocol, acting as a unique identifier for the device on the network. The function updates the Ethernet controller's PALR (Physical Address Lower Register) and PAUR (Physical Address Upper Register) with the provided MAC address.

```
void EnetSetMacAddress(const uint8_t * address)
{
    ENET->PALR = (((uint32_t) address[0]) << 24) |
                 (((uint32_t) address[1]) << 16) |
                 (((uint32_t) address[2]) << 8) |
                 (((uint32_t) address[3]) << 0);
    ENET->PAUR = (((uint32_t) address[4]) << 24) |
                 (((uint32_t) address[5]) << 16);
}
```

8.6 Enabling ENET

This function enables the Ethernet operation based on the link status and speed settings provided by the PHY (Physical Layer). It configures the controller to match the link's duplex and speed settings, ensuring that the Ethernet controller communicates correctly over the network.

```
int EnetEnable(const MiimStatus_T * status)
{
    if (status->linkUp)
    {
        switch (status->mode.duplex)
        {
            case LinkDuplexHalf:
                ENET->RCR |= ENET_RCR_DRT_MASK;
                ENET->TCR &= ~ENET_TCR_FDEN_MASK;
                break;
            case LinkDuplexFull:
                ENET->RCR &= ~ENET_RCR_DRT_MASK;
                ENET->TCR |= ENET_TCR_FDEN_MASK;
                break;
            default:
                return EPERM;
        }
        switch (status->mode.speed)
        {
            case LinkSpeed10Mbps:
                ENET->RCR |= ENET_RCR_RMII_10T_MASK;
                break;
            case LinkSpeed100Mbps:
                ENET->RCR &= ~ENET_RCR_RMII_10T_MASK;
                break;
            default:
                return EPERM;
        }

        EnetSetup();

        return 0;
    }
    else
    {
        return EPERM;
    }
}
```

Disabling ENET

This function disables the Ethernet controller and is useful for power management or when reconfiguring the network settings requires the Ethernet controller to be temporarily turned off. It is written to ensure that no data transmission or reception occurs until the Ethernet controller is explicitly re-enabled.

```
void EnetDisable()
{
    ENET->EIMR = 0;                    /* mask all interrupt sources */
    ENET->EIR = 0x7FFF8000;            /* acknowledge any pending interrupts */
    ENET->ECR &= ~ENET_ECR_ETHEREN_MASK;            /* disable the controller */
}
```

8.7 Data Transmission

There are two functions written to send data over the network, **EnetGetTxBuffer** and **EnetSendTxBuffer**. **EnetGetTxBuffer** retrieves a pointer to the next available transmit buffer where data can be written, and once the data is written into the buffer, **EnetSendTxBuffer** is called to mark the buffer for transmission. These functions manage the buffer descriptors, ensuring that data is correctly queued for sending out on the network.

```
uint8_t * EnetGetTxBuffer()
{
    /* block if no buffers are available */
    xSemaphoreTake(EnetData.txSem, OS_WAIT_FOREVER);
    return EnetData.txBufferDescriptor[EnetData.nextTxBuffer].buffer;
}

void EnetSendTxBuffer(uint16_t len)
{
    /* setup the buffer descriptor for transmission */
    EnetData.txBufferDescriptor[EnetData.nextTxBuffer].length = len;
    EnetData.txBufferDescriptor[EnetData.nextTxBuffer].control |=
ENETTXBUF_CONTROL_R | ENETTXBUF_CONTROL_L;

    /* manage buffer wrap-around */
    EnetData.nextTxBuffer++;
    if (EnetData.nextTxBuffer>=ENET_TX_BUFFER_NUMBER)
    {
        EnetData.nextTxBuffer = 0;
    }

    /* indicate that the transmit descriptor ring has been updated */
    ENET->TDAR = ENET_TDAR_TDAR_MASK;
}
```

8.8 Data Reception

Two functions are created for data reception, **EnetRecvAt** and **EnetFreeRxBuffer**. **EnetRecvAt** allows access to the received data at a specified offset within the current receive buffer, while **EnetFreeRxBuffer** releases the receive buffer back to the pool, making it available for future packets. Together, they manage the reception process, allowing the application to process incoming data and then prepare the buffer for new data.

```
uint8_t * EnetRecvAt(uint16_t len)
{
    return &(EnetData.rxBufferDescriptor[EnetData.nextRxBuffer].buffer[len]);
}

void EnetFreeRxBuffer()
{
    /* free the buffer descriptor after reception */
    EnetData.rxBufferDescriptor[EnetData.nextRxBuffer].control |=
ENETRXBUF_CONTROL_E;

    /* indicate that the receive descriptor ring has been updated */
    ENET->RDAR = ENET_RDAR_RDAR_MASK;

    /* manage buffer wrap-around */
    EnetData.nextRxBuffer++;
    if (EnetData.nextRxBuffer>=ENET_RX_BUFFER_NUMBER)
    {
        EnetData.nextRxBuffer = 0;
    }
}
```

9 Firmware Testing in Loopback Mode

9.1 Loopback Mode

Loopback mode is a diagnostic feature implemented in various network devices to facilitate testing and troubleshooting. This mode "loops" the transmitted signals back to the receiver within the same device, and allows for the verification of the transmit and receive paths of the device, ensuring that both are functioning correctly.

In loopback mode, data packets sent by the transmitting part of the device are redirected internally to the receiving part of the device. The Ethernet controller in the S32K344 board is configured to activate loopback by configuring specific registers within the Ethernet MAC (Media Access Controller).

9.2 Firmware Testing

The developed EMAC driver was tested for functionality and reliability using loopback mode to verify the firmware's capability to handle Ethernet data packets.

9.2.1 Objectives of Loopback Testing

The primary objective of the loopback test was to validate the EMAC driver by sending and receiving any data string, and a specific string was used - "**Hello Haris Khan**". This test aimed to demonstrate the driver's ability to:

- Initialize the EMAC hardware into loopback mode.
- Transmit data packets from the microcontroller to the EMAC.
- Receive the transmitted data internally without external network involvement.
- Verify the integrity and content of the received data.

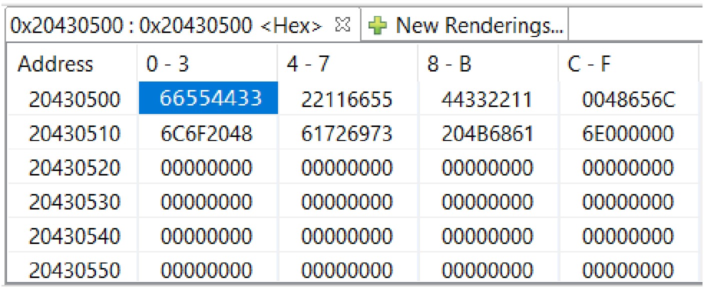
9.2.2 Test Setup

The test setup involved configuring the EMAC in hardware loopback mode, where packets transmitted by the EMAC are internally routed back to the receiver. This setup eliminates external factors that could affect the test, such as network congestion or physical layer issues, allowing for focused testing of the driver and EMAC hardware.

Initialization: The microcontroller, hosting the EMAC driver, was initialized, and the EMAC hardware was configured into loopback mode.

Data Preparation: A packet containing the string "**Hello Haris Khan**" was prepared for transmission. This packet was structured according to Ethernet protocol standards, including the necessary headers.

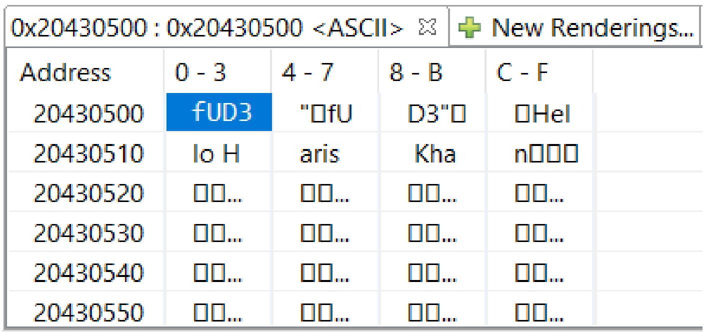
Transmission: The prepared packet was transmitted using the EMAC driver. The driver writes the string into the **TX Ring Data Buffer**, which is present at the address **0x20430500.**


0x20430500 : 0x20430500 <Hex> New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
20430500	66554433	22116655	44332211	0048656C
20430510	6C6F2048	61726973	204B6861	6E000000
20430520	00000000	00000000	00000000	00000000
20430530	00000000	00000000	00000000	00000000
20430540	00000000	00000000	00000000	00000000
20430550	00000000	00000000	00000000	00000000

Figure 9.1: Memory contents showing "Hello Haris Khan" in TX Ring Data Buffer in Hex

The TX Ring Data Buffer can also be viewed below in ASCII.


0x20430500 : 0x20430500 <ASCII> New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
20430500	fUD3	"□fU	D3"□	□Hel
20430510	lo H	aris	Kha	n□□□
20430520	□□...	□□...	□□...	□□...
20430530	□□...	□□...	□□...	□□...
20430540	□□...	□□...	□□...	□□...
20430550	□□...	□□...	□□...	□□...

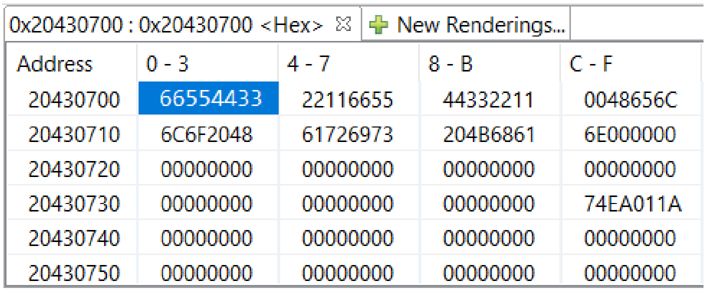
Figure 9.2: Memory contents showing "Hello Haris Khan" in TX Ring Data Buffer in ASCII

<u>Reception:</u> The EMAC hardware, still in loopback mode, receives the packet and sends it to the **RX Ring Data Buffer**, present in memory at the address **0x20430700**.

<u>Verification:</u> The received packet was inspected to verify that the data content matched the transmitted string "**Hello Haris Khan**". This verification process involved comparing the received data with the expected string at the memory level, ensuring that the packet had been correctly transmitted and received without any alterations or loss.

9.2.3 <u>Test Results</u>

The loopback test was successful, with the "**Hello Haris Khan**" string being correctly transmitted and received by the EMAC driver. The integrity of the data was confirmed by inspecting the memory location of the **RX Ring Data Buffer** where the received packet was stored.


0x20430700 : 0x20430700 <Hex> New Renderings...

Address	0 - 3	4 - 7	8 - B	C - F
20430700	66554433	22116655	44332211	0048656C
20430710	6C6F2048	61726973	204B6861	6E000000
20430720	00000000	00000000	00000000	00000000
20430730	00000000	00000000	00000000	74EA011A
20430740	00000000	00000000	00000000	00000000
20430750	00000000	00000000	00000000	00000000

Figure 9.3: Memory contents showing "Hello Haris Khan" in RX Ring Data Buffer in Hex

10 Conclusion

This thesis presents a comprehensive analysis and development of an Ethernet driver, focusing on the S32K344 microcontroller, within the context of automotive Ethernet applications. The S32K344, part of NXP's S32K3 family of microcontrollers, is designed to meet the demanding requirements of automotive applications, offering enhanced features for reliable communication and control. Our work has focused on exploiting these capabilities to develop a robust Ethernet driver that supports high-speed data transmission and reception, essential for modern automotive networks. This work is situated within the broader effort to enhance vehicular networks' efficiency and reliability, critical for supporting advanced features such as autonomous driving, infotainment systems, and vehicle-to-everything (V2X) communications.

The development process involved a detailed examination of the Ethernet protocol as implemented on the S32K344, ensuring that the driver is optimized for the microcontroller's architecture and the specific needs of automotive Ethernet communication. This entailed a rigorous analysis of the hardware and software interfaces, the network stack, and the integration points with the microcontroller's peripherals and processing units.

A critical part of our development and testing phase was the utilization of the driver in loopback mode, which proved to be a pivotal step in validating the functionality and reliability of the Ethernet communication. This mode allowed for the successful transmission and reception of data, demonstrating the driver's capability to handle Ethernet frames efficiently and accurately without the need for an external network. This testing phase was instrumental in identifying and rectifying potential issues, ensuring a high level of reliability and performance.

To further extend the applicability and robustness of our Ethernet driver, future work will focus on more complex network configurations involving multiple ECUs. This will involve the use of sophisticated testing tools like Vector CANoe, which will enable the simulation and analysis of network behaviors, traffic patterns, and performance metrics in a controlled environment. Such testing is crucial for validating the driver's performance in real-world automotive network scenarios, where reliability and data integrity are paramount.

The development and debugging of the Ethernet driver were significantly aided by the use of a PE Micro board, which facilitated a hands-on approach to testing and verification. This tool was invaluable for real-time debugging and for making iterative improvements to the driver code, ensuring optimal performance and compatibility with the S32K344 microcontroller.

An essential aspect of this project was the alignment with the AUTOSAR software architecture, particularly in developing the kernel and configuration for the Ethernet driver. This adherence to AUTOSAR standards not only ensures that the driver is compatible with a wide range of automotive software systems but also underscores the importance of standardized approaches in the development of automotive software. By integrating our Ethernet driver within the basic software layer of AUTOSAR, we have laid a foundation for future enhancements and broader applicability in automotive networks.

In conclusion, this thesis has laid the groundwork for advanced Ethernet communication in automotive applications, leveraging the capabilities of the S32K344 microcontroller. It underscores the critical role of Ethernet in transforming automotive networks, paving the way for more connected and intelligent vehicles.