

Article

Development of a Common API for Multiple Ethernet Fieldbus Protocols in Embedded Slave Devices

Donghyuk Kim  and Joon-Young Choi * 

Department of Electrical and Electronic Engineering, Pusan National University, Busan 46241, Republic of Korea; rubidius@pusan.ac.kr

* Correspondence: jyc@pusan.ac.kr

Abstract: Slave devices in Ethernet-based fieldbus networks often require extensive re-programming of applications and replacement of protocol stacks and Ethernet drivers whenever the fieldbus protocol needs to be changed. To address this challenge, we develop a common application programming interface (API) and stack interfaces that enable seamless protocol switching among EtherCAT, PROFINET, and EtherNet/IP without requiring protocol-specific code modifications. The real-time data exchange between the API and each protocol stack is realized in the stack interface by using the synchronization mechanism provided by FreeRTOS. The developed common API and stack interfaces facilitate the development of slave device applications that are universally compatible with multiple protocols, EtherCAT, PROFINET, and EtherNet/IP. Moreover, once a required protocol is selected in the integrated development environment (IDE) software before building the slave device firmware, the corresponding protocol stack and Ethernet drivers are automatically specified and the need to replace protocol stacks or Ethernet drivers is even eliminated when switching protocols. To validate the developed common API and stack interfaces, they were implemented on a slave device using TI's TMDS243EVM board, and a fieldbus network was built by connecting the slave device to a master device executed by Beckhoff's TwinCAT on a Windows PC. Experimental results confirmed the API's functionality, reliability, and practical applicability in streamlining protocol management for Ethernet-based fieldbus networks.



Academic Editor: Christos J. Bouras

Received: 30 December 2024

Revised: 30 January 2025

Accepted: 1 February 2025

Published: 5 February 2025

Citation: Kim, D.; Choi, J.-Y.

Development of a Common API for Multiple Ethernet Fieldbus Protocols in Embedded Slave Devices.

Electronics **2025**, *14*, 613.<https://doi.org/10.3390/electronics14030613>**Copyright:** © 2025 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license

<https://creativecommons.org/licenses/by/4.0/>.

Keywords: common API; Ethernet-based fieldbus protocols; EtherCAT; PROFINET; EtherNet/IP; slave embedded system; industrial communication

1. Introduction

Industry 4.0 seeks to revolutionize manufacturing by integrating information and communication technology (ICT) with advanced production technologies to create smart factories. Significant progress in research and technological development has been made to realize this vision [1,2]. A robust and reliable communication system that enables seamless data exchange is fundamental to implementing Industry 4.0 in manufacturing environments. Such communication systems play a pivotal role in integrating digital and physical processes, forming the foundation of smart manufacturing [3,4].

In smart factories, communication protocols that enable real-time data exchange and ensure interoperability among diverse devices and systems play a crucial role. This perspective has also been emphasized in previous studies. It is highlighted in [5] that the integration of widely supported open protocols is essential for managing the interoperability challenges inherent in industrial Internet of things (IIoT) environments. Additionally,

with the ongoing convergence of operation technology (OT) and information technology (IT), industrial networks face the challenge of efficiently managing diverse and critical data. To support Industry 4.0 initiatives, these networks must be designed with enhanced flexibility and reliability to meet evolving demands [6].

Industrial Ethernet-based fieldbus protocols, such as EtherCAT, PROFINET, and EtherNet/IP, have increasingly supplanted traditional fieldbus protocols in recent years. This transition is driven by their cost-effectiveness and seamless integration capabilities across all enterprise levels. These Ethernet-based fieldbus protocols deliver high reliability, real-time communication, and enhanced production efficiency and flexibility, establishing them as essential enablers of intelligent manufacturing [7–10].

A key consideration for smart factories is their construction with a complex mix of equipment and devices from various manufacturers, often utilizing different industrial Ethernet-based protocols [11]. In this situation, the development of factory devices capable of supporting multiple protocols is crucial to ensuring compatibility and operational flexibility. Multi-protocol supporting devices simplify system integration, improve operational efficiency, and lower both development and management costs. Moreover, they enhance product competitiveness by addressing the diverse requirements of users, making them an essential component of smart factory systems [12–15].

However, even when device hardware supports multiple protocols, significant software effort is required to switch between them. Each time a protocol is changed, the device firmware must be rebuilt. This process involves replacing the protocol stack and Ethernet driver in the project environment and reprogramming the application for the newly selected protocol. Consequently, separate software must be implemented and maintained for each protocol, leading to increased maintenance complexity and higher costs. Moreover, whenever protocols are replaced, an additional challenge lies in installing and testing cybersecurity protection features to meet the security requirements of each protocol stack.

To address the challenges associated with supporting multiple protocols, we develop a common application programming interface (API) and stack interfaces that enable seamless protocol switching among three Ethernet-based fieldbus protocols, EtherCAT, PROFINET, and EtherNet/IP, which hold the highest market share in the Ethernet-based industrial fieldbus protocol sector. The API provides a unified set of functions for writing application programs compatible with all three protocols. Moreover, in the common API environment, cybersecurity protections for each protocol stack can be pre-applied and rigorously tested, requiring only the applications to be rewritten when protocols are replaced. By using the synchronization mechanism provided by FreeRTOS, the stack interface ensures real-time data exchange between the application and each protocol stack.

This development facilitates efficient protocol switching for slave devices, significantly reducing the time and cost associated with developing and maintaining communication devices for smart factories. Notably, the developed common API future-proofs installations by enabling slave devices to remain unchanged while only the master device requires replacement when switching to a different protocol. This protocol switching approach appreciably reduces migration efforts and hardware replacement costs in particular. The functionality and reliability of the developed common API and stack interfaces are validated through extensive experiments conducted on an embedded board powered by TI's AM2434 microcontroller unit (MCU).

The organization of this paper is as follows: Section 2 provides a brief introduction to key background knowledge on Ethernet-based fieldbus protocols, including EtherCAT, PROFINET, and EtherNet/IP. Section 3 describes the design and implementation of the common API and stack interfaces. In Section 4, a fieldbus network comprising a PC and an embedded board is constructed, and experiments are conducted to verify the

functionality of the developed common API and stack interfaces. Finally, Section 5 presents the concluding remarks.

2. Key Background Knowledge

A general industrial network structure is depicted in Figure 1, illustrating the hierarchical organization spanning corporate, control, and fieldbus networks [16]. As shown in Figure 1, the corporate network level comprises office applications, data servers, and back-office servers running enterprise resource planning (ERP), manufacturing execution systems (MESs), and other enterprise-level software. The control network level consists of programmable logic controllers (PLCs), PC-based or embedded controllers, and human-machine interfaces (HMIs). The fieldbus network level consists of field devices such as actuators, sensors, I/O devices, and robots.

The corporate network may be a plantwide local area network (LAN) linking office computers for enterprise management. It connects seamlessly with the control network to provide enterprise-level insights and support data-driven decision-making. The control network acts as an intermediary between corporate and fieldbus networks, facilitating real-time data exchange. It ensures that automation controllers and HMIs effectively utilize information from field devices in supervisory applications such as supervisory control and data acquisition (SCADA) or distributed control systems (DCSs). At the lowest level, the fieldbus network ensures the reliable transmission of process, diagnostic, and configuration data between field devices and controllers. This communication supports the efficient operation of industrial equipment while maintaining time synchronization for coordinated activities.

The developed common API and stack interface running on the slave devices enable real-time data exchange with the master device based on Ethernet-based fieldbus protocols. This operation is conducted at the fieldbus network level depicted in Figure 1. In the following subsections, we briefly introduce the key background knowledge on three Ethernet-based fieldbus protocols—EtherCAT, PROFINET, and EtherNet/IP—that will be integrated into the developed common API.

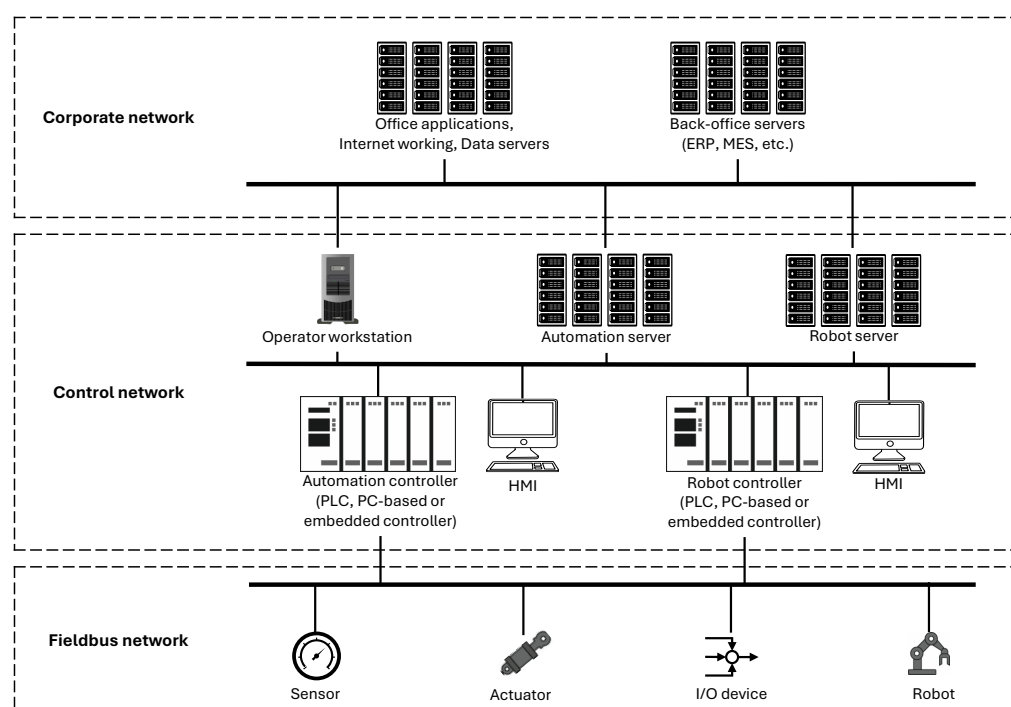


Figure 1. Structure of general industrial network.

2.1. EtherCAT

EtherCAT is an open real-time Ethernet network protocol originally developed by Beckhoff Automation [17]. It achieves short cycle times and high bandwidth through an “on-the-fly” data exchange mechanism, allowing the master and multiple slave devices to communicate efficiently within the short duration of a single Ethernet frame transmission [18].

- **Device Model**

EtherCAT organizes devices using an object dictionary and Process Data Objects (PDOs), providing a logical structure for device communication. The object dictionary, based on the CANopen standard, serves as a database that organizes all the parameters and variables of a slave device. Each parameter and variable is uniquely identified by an index and subindex, enabling the master to interpret and control the slave's functions and status.

PDOs facilitate real-time process data exchange between the master and slave devices. These data structures enable periodic data transfers, ensuring high-performance real-time communication. The PDO mapping specifies which data are transferred as process data, simplifying data mapping and parameter management in the slave implementation.

The EtherCAT Slave Information (ESI) file, provided by the device manufacturer, contains the device's specifications, communication capabilities, and configuration parameters, which are used by the master configuration software. The ESI file contains the object dictionary structure and possible PDO configuration options of the slave, which the master uses to identify the slave and establish the appropriate PDO mapping during initialization. By accessing the slave's object dictionary, the master can read and write parameters, facilitating efficient setup and integration of the automation system.

- **Connection Configuration**

EtherCAT uses Service Data Objects (SDOs) and PDOs as core concepts to establish and manage the communication relationship between master and slave devices.

- **PDO**

PDOs are used for periodic process data exchange, ensuring high-performance real-time communication. The master retrieves PDO mapping information from the slave's ESI file and configures communication, enabling seamless real-time data exchange.

- **SDO**

SDOs handle aperiodic data exchange and parameter settings between the master and slave. Through SDOs, the master can access the slave's object dictionary to read parameters and modify writable parameters, which is crucial for device configuration and diagnostics.

The master initiates communication by sending SDO requests and manages the slave's configuration and parameters. This process ensures reliable and stable communication while simplifying system maintenance and diagnostics.

2.2. PROFINET

PROFINET is an industrial Ethernet standard developed by PROFIBUS International (PI), designed for modular machine and plant engineering as well as distributed I/O systems [19]. To optimize real-time performance and reduce the computational burden on controllers, PROFINET IO (PNIO) simplifies the protocol stack by removing the IP layer. Instead, it employs a middleware layer above the Ethernet MAC layer to enable high-performance data transfer, cyclic data exchange, and event-driven signal transmission [20].

- **Device Model**
PROFINET organizes devices through a structured device model, enabling modular and efficient system organization. Devices are divided into slots and subslots, where each slot corresponds to a module that provides specific functions or data. Subslots represent sub-functional units within a module, further refining the organization. Each slave device communicates its functional capabilities, parameters, and communication settings to the master using a General Station Description (GSD) file, provided by the device manufacturer. This file simplifies the configuration and integration of the automation system by providing essential information for system setup.
- **Connection Configuration**
PROFINET introduces the concept of an Application Relation (AR) to establish and manage communication between master and slave devices. AR serves as a logical communication channel that ensures reliable and real-time data exchange. It is categorized into three types:
 - IO AR: manages I/O data exchange.
 - Supervisor AR: handles system diagnostics and maintenance.
 - Alarm AR: facilitates alarm transmission.

The master initiates communication by sending a connect request to the slave, establishing an AR based on the configured parameters. This mechanism ensures efficient operation and real-time communication across the network.

2.3. EtherNet/IP

EtherNet/IP is an adaptation of the Common Industrial Protocol (CIP) to the TCP/IP suite, built on the IEEE 802.3 Ethernet standard [21]. Unlike EtherCAT and PROFINET, EtherNet/IP employs IP protocols for real-time data exchange, leveraging CIP as its application layer [22].

- **Device Model**
EtherNet/IP structures devices using the class and object concepts of CIP. In CIP, a device is represented as a collection of objects, each providing specific functions or data. These objects are instances of corresponding classes and are defined by their attributes, services, and data. EtherNet/IP devices communicate their specifications, communication capabilities, and configuration parameters to the master using an Electronic Data Sheet (EDS) file, provided by the device manufacturer. The EDS file describes device classes, objects, and attributes, facilitating the setup and integration of automation systems by providing the master with detailed information for configuration.
- **Connection Configuration**
In EtherNet/IP, the Connection Manager Object is responsible for establishing and managing CIP connections, which handle real-time data communication. The master initiates a CIP connection by sending a connection request to the device. Each CIP connection is uniquely identified by a Connection ID and is associated with specific attributes. These attributes allow the system to monitor the connection status and configure communication parameters effectively.
This structure enables EtherNet/IP to organize communication between devices, efficiently manage real-time data exchange, and provide diagnostic capabilities. By leveraging CIP's class and object model, EtherNet/IP creates logical communication channels that ensure reliability and real-time communication, making it an effective solution for industrial automation systems.

3. Design and Implementation

In this section, we design and implement the proposed common API and stack interfaces for the efficient, immediate, and seamless protocol replacement among the industrial Ethernet-based fieldbus protocols, EtherCAT, PROFINET, and EtherNet/IP. It is noted that the design and implementation assume fieldbus networks consisting of a master and a single slave but can be applied to networks consisting of a master and multiple slaves without requiring any modifications.

3.1. Hardware Selection

In this subsection, an appropriate MCU and embedded system board are selected to implement the proposed common API and stack interfaces to support multiple Ethernet-based fieldbus protocols. TI's AM2434 MCU is selected for its robust support of industrial Ethernet protocols, including EtherCAT, PROFINET, and EtherNet/IP. Central to its functionality is TI's Programmable Real-time Unit and Industrial Communication Subsystem (PRU-ICSS), a specialized hardware subsystem designed to handle the physical and data link layers of these protocols.

The PRU-ICSS includes compact, efficient processor cores (PRUs), shared data and instruction memories, internal peripheral modules, and an interrupt controller (INTC). This subsystem is specifically designed to ensure deterministic latencies in packet communication by preventing packet collisions, making it highly suitable for industrial Ethernet protocol implementations [23]. Its flexibility enables the software-based implementation of the physical and data link layers of major industrial Ethernet protocols, allowing the seamless replacement of protocol stacks as needed.

The AM2434 also supports FreeRTOS, an open-source real-time operating system. FreeRTOS is designed to be lightweight and portable with optimizations for low memory usage and fast context switching. To streamline development, TI provides a comprehensive Software Development Kit (SDK) (09.02.00) that includes optimized drivers and libraries for the FreeRTOS environment.

For this study, TI's TMDS243EVM development board is chosen as the hardware platform. The TMDS243EVM is a standalone test and prototyping platform specifically designed for evaluating the functionality of the AM243x MCU series. Its compatibility with industrial Ethernet protocols and its suitability for embedded system development make it an ideal choice for implementing a common API and stack interfaces for multiple Ethernet fieldbus protocols in AM2434-based systems.

3.2. Overall Structure of Integrated Protocols

Using the developed common API and stack interfaces, the three protocols—EtherCAT, PROFINET, and EtherNet/IP—are integrated to operate under a unified API. The overall structure of the integrated protocols is illustrated in Figure 2. As illustrated in Figure 2, the physical and link layers of each protocol are implemented using TI's PRU-ICSS. The stack interface of each protocol is implemented based on the synchronization mechanism provided by FreeRTOS.

To facilitate the selection of the protocol required for a specific application, a pop-up menu is implemented using the build configuration functionality within Code Composer Studio (CCS), TI's integrated development environment (IDE) for its MCUs. This pop-up menu provides a user-friendly interface where developers can choose the desired protocol among EtherCAT, PROFINET, or EtherNet/IP. Once a protocol is selected through the pop-up menu, CCS automatically updates the build settings to include the appropriate protocol stack, Ethernet driver, and related configurations. This streamlined process ensures that

the generated firmware aligns precisely with the requirements of the selected protocol, minimizing manual intervention and enhancing development efficiency.

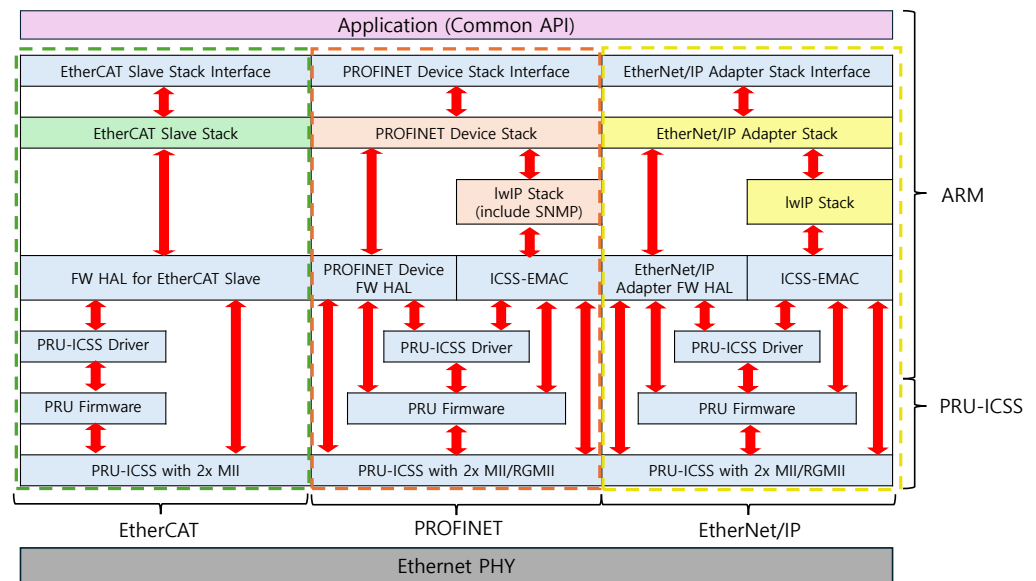


Figure 2. Overall structure of integrated protocols.

3.3. EtherCAT Integration

The EtherCAT physical and link layers are implemented using TI's PRU-ICSS along with its PRU firmware. The EtherCAT slave stack code is generated using the EtherCAT Slave Stack Code (SSC) tool V5.13 provided by the EtherCAT Technology Group (ETG). To enable seamless operation of the EtherCAT protocol through a unified API, it is necessary to design and implement both a suitable EtherCAT stack interface and the common API. The EtherCAT stack interface plays a critical role in facilitating real-time data exchange, which is achieved by developing multiple tasks synchronized with process data reception. This synchronization is effectively managed using the FreeRTOS synchronization mechanism. In the following, we detail the design and implementation of the stack interface layer and the common API layer for EtherCAT integration.

3.3.1. EtherCAT Stack Interface Layer

For the EtherCAT stack interface, two FreeRTOS tasks and two functions are designed and implemented to manage process data communication within an EtherCAT slave device. This interface handles incoming data from the EtherCAT master by storing them from the EtherCAT slave controller (ESC), implemented on the PRU-ICSS, to the input variables of the stack interface. Conversely, this interface transfers outgoing data, which are prepared and stored in the output variables of the stack interface by the application program, back to the ESC. The ESC subsequently transmits the outgoing data to the EtherCAT master. In the following, we detail the roles and functionalities of the developed FreeRTOS tasks and functions for the EtherCAT stack interface layer, emphasizing their design and implementation for seamless EtherCAT protocol integration.

- Sync0task

The Sync0task is triggered by the ESC upon the generation of a Sync0 interrupt. The Sync0 interrupt, derived from the ESC's distributed clock, ensures accurate synchronization between the slave application tasks and the cyclical operations of the network. When the Sync0 interrupt is issued, the Sync0task is executed, enabling the slave tasks to remain precisely aligned with the network's operation cycle.

During its execution, the Sync0task runs the application program in the common API layer, where outgoing data are prepared and stored in the output variables of the stack interface. Subsequently, the Sync0task calls the APPL_InputMapping function to transfer the outgoing data to the ESC, which subsequently transmits them to the EtherCAT master, ensuring seamless real-time communication. It is noted that the Sync0task is responsible exclusively for data transmission and does not handle data reception within the stack interface layer. The overall function call sequence and the outgoing data flow managed by the Sync0task are illustrated in Figure 3.

- **PDItask**
The PDItask is triggered by the ESC upon the generation of a Process Data Interface (PDI) interrupt. This interrupt is issued by the ESC to indicate that incoming data are ready for reception. Upon the issuance of the PDI interrupt, the PDItask is executed. During its execution, the PDItask invokes the APPL_OutputMapping function in the EtherCAT stack interface, which transfers the incoming data from the ESC to the input variables (ECAT_recvContainer), defined in the stack interface layer. Following this, the PDItask runs the application program in the common API layer, where the application retrieves and processes the incoming data. The application program then prepares the outgoing data and stores them in the output variables (ECAT_sendContainer), also defined in the stack interface layer. Subsequently, the PDItask invokes the APPL_InputMapping function to transfer the outgoing data to the ESC, which transmits them to the EtherCAT master, ensuring seamless real-time communication.
Notably, unlike the Sync0task, the PDItask handles both data reception and transmission within the stack interface layer. The overall function call sequence and the data flow managed by the PDItask are illustrated in Figures 3 and 4.
- **APPL_OutputMapping function**
When the incoming data from the EtherCAT master are received, this function is invoked by the PDItask to update the input variables (ECAT_recvContainer) in the stack interface with the newly received data. Specifically, this function retrieves the incoming data from the master, stored in the ESC, and transfers them to the input variables (ECAT_recvContainer) in the stack interface, making them accessible for processing by the application program. It is noted that the term “Output” used in the function name means that the function processes data that are output from the master’s perspective but are input from the slave’s perspective.
- **APPL_InputMapping function**
When the application program prepares and stores outgoing data intended for the EtherCAT master in the output variables (ECAT_sendContainer) of the stack interface, the PDItask or Sync0task invokes this function to transfer the outgoing data from the output variables (ECAT_sendContainer) to the ESC. The ESC then transmits the data to the EtherCAT master, ensuring seamless real-time communication. It is noted that the term “Input” used in the function name means that the function processes data that are input from the master’s perspective but are output from the slave’s perspective.

3.3.2. Common API Layer for EtherCAT

For the EtherCAT common API layer, four functions are implemented to provide a high-level interface for managing communication and operations within the EtherCAT slave device. These functions facilitate seamless application program development using the EtherCAT protocol stack. The developed API functions simplify process data exchange and task execution, ensuring efficient and reliable real-time communication. Notably, these functions are designed to be commonly utilized for the PROFINET and EtherNet/IP protocols as well. In the following, we detail the roles and functionalities of

the developed functions for the EtherCAT common API layer, emphasizing their design and implementation for seamless EtherCAT protocol integration.

- **CommonAPI_Init function**
This function initializes the hardware and protocols of the slave device while also creating the necessary FreeRTOS tasks.
- **CommonAPI_Application function**
This function acts as a wrapper, invoked by the Sync0task or PDItask of the EtherCAT stack interface layer, to execute the slave application program in the common API layer, as illustrated in Figures 3 and 4.
- **CommonAPI_RecvData function**
This function is designed to handle the reception of incoming data on a per-byte basis. It retrieves the incoming data from the input variables (ECAT_recvContainer) of the EtherCAT stack interface that are storing the incoming data transferred from the ESC via the APPL_OutputMapping function of the stack interface.
In the experimental scenario described in Section 4, a total of 32 bytes of incoming data from the master to the slave is predefined. These incoming data are organized into eight 4-byte variables, which are labeled as Output1 to Output8 of EtherCAT process data. The function accepts a single argument, recvIndex, which specifies the byte index of the received process data. For example, invoking CommonAPI_RecvData(10) returns the value of the second byte of Output2, corresponding to the 10th byte within the 32 bytes of incoming data from the master. Figure 4 illustrates the data flow when this function is executed.
- **CommonAPI_SendData function**
This function is designed to handle the transmission of outgoing data on a per-byte basis. It writes the outgoing data values to the output variables (ECAT_sendContainer) of the EtherCAT stack interface, which will be transferred to the ESC via the APPL_InputMapping function of the stack interface.
In the experimental scenario described in Section 4, a total of 32 bytes of outgoing data from the slave to the master is predefined. These outgoing data are organized into eight 4-byte variables, which are labeled as Input1 to Input8 of EtherCAT process data. The function requires two arguments: sendData, representing the value to be sent, and sendIndex, which specifies the byte index of the process data to be transmitted. For example, calling CommonAPI_SendData(20, 30) sets 20 as the value of the second byte of Input8, corresponding to the 30th byte within the 32 bytes of outgoing data to be sent to the master. Figure 3 illustrates the data flow when this function is executed.

Table 1 provides a list of all functions and FreeRTOS tasks developed for the common API and EtherCAT stack interface to enable efficient data communication in the EtherCAT protocol.

Table 1. Functions and FreeRTOS tasks developed for the common API and stack interface of the EtherCAT protocol.

Layer	Function	FreeRTOS Task
EtherCAT stack interface	APPL_OutputMapping	Sync0task
	APPL_InputMapping	PDItask
Common API	CommonAPI_Init	
	CommonAPI_Application	
	CommonAPI_RecvData	
	CommonAPI_SendData	

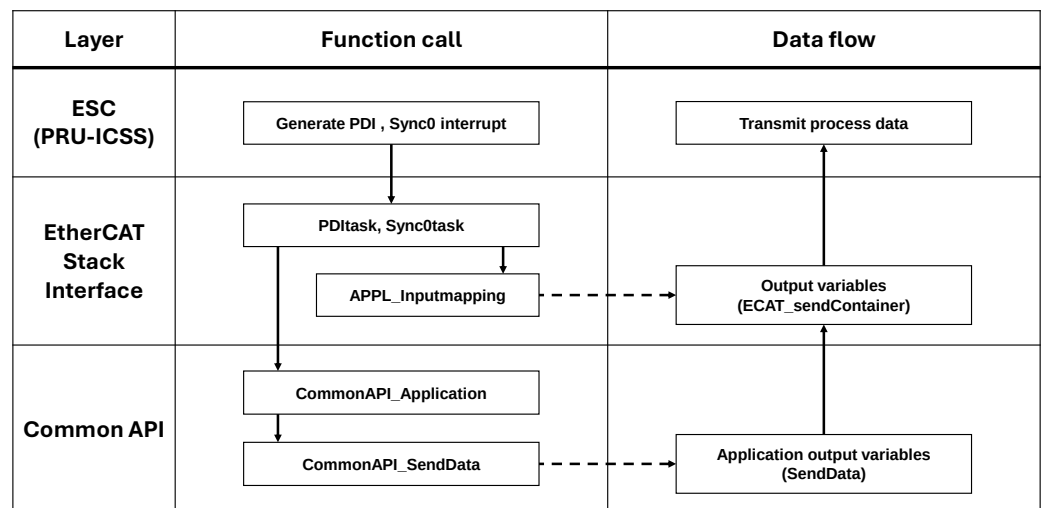


Figure 3. Function call procedure and data flow during the outgoing data transmission in the EtherCAT protocol.

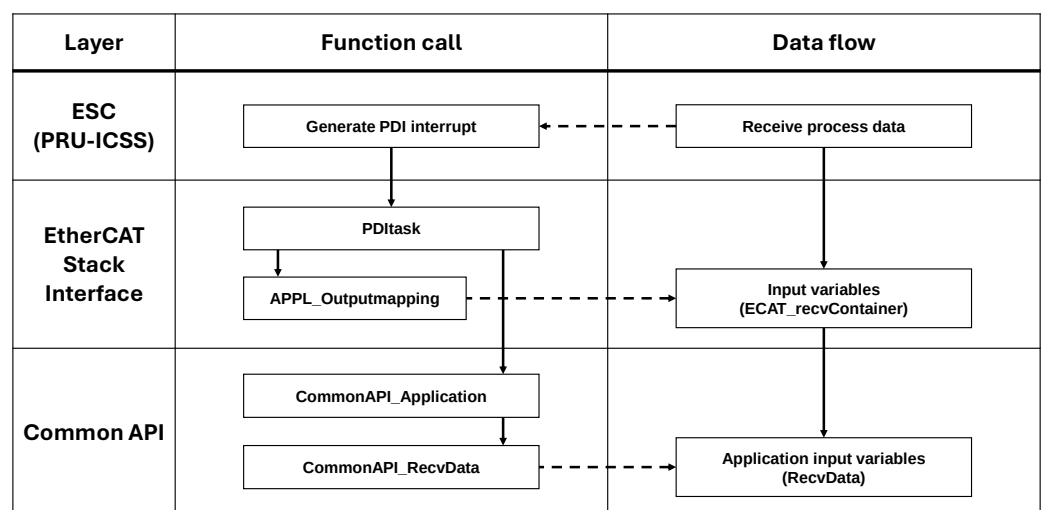


Figure 4. Function call procedure and data flow during the incoming data reception in the EtherCAT protocol.

3.4. PROFINET Integration

Similar to the EtherCAT protocol, the PROFINET physical and link layers are implemented using TI's PRU-ICSS along with its PRU firmware. The PROFINET slave stack is obtained from the TI's Industrial Communications SDK for AM243x (09.02.00.15). To enable seamless operation of the PROFINET protocol through a unified API, it is necessary to design and implement both a suitable PROFINET stack interface and a common API. The PROFINET stack interface plays a critical role in facilitating real-time data exchange, which is achieved by developing a FreeRTOS task synchronized with incoming data reception. This synchronization is effectively managed using the FreeRTOS synchronization mechanism. In the following, we detail the design and implementation of the stack interface layer and the common API layer for PROFINET integration.

3.4.1. PROFINET Stack Interface Layer

For the PROFINET stack interface, a FreeRTOS task and two functions are implemented to manage process data communication within a PROFINET slave device. The PROFINET stack interface processes incoming data from the PROFINET master by storing them from the PRU-ICSS to the input variables of the stack interface. Conversely, this

interface transfers outgoing data, which are prepared and stored in the output variables of the stack interface by the application program, back to the PRU-ICSS. Subsequently, the PROFINET link and physical layer implemented on PRU-ICSS transmits the outgoing data to the PROFINET master. In the following, we detail the roles and functionalities of the developed FreeRTOS task and functions for the PROFINET stack interface layer, emphasizing their design and implementation for seamless PROFINET protocol integration.

- PN_APP_IOD_cyclicDataTask**
 The PN_APP_IOD_cyclicDataTask is triggered by a PRU-ICSS interrupt when the PRU-ICSS receives incoming process data from the master. Upon activation, the task invokes the PN_APP_IOD_cbDataRead function in the PROFINET stack interface and stores the incoming data from the PRU-ICSS into the input variables (PN_rcvContainer) defined in the stack interface layer. Following this, the task executes the application program in the common API layer, where the incoming data are retrieved and processed. The application program then prepares the outgoing data and stores them in the output variable (PN_sendContainer), also defined in the stack interface layer. Subsequently, the PN_APP_IOD_cyclicDataTask invokes the PN_APP_IOD_cbDataWrite function to transfer the outgoing data to the PRU-ICSS, which transmits them to the PROFINET master, ensuring seamless real-time communication. The overall function call sequence and the data flow managed by the PN_APP_IOD_cyclicDataTask are illustrated in Figures 5 and 6.
- PN_APP_IOD_cbDataRead function**
 When incoming data are received from the PROFINET master, this function is invoked by the PN_APP_IOD_cyclicDataTask to update the input variables (PN_rcvContainer) in the stack interface with the newly received data. Specifically, this function retrieves the incoming data from the master, stored in the PRU-ICSS, and stores them to the input variables (PN_rcvContainer) in the stack interface, making them accessible for processing by the application.
- PN_APP_IOD_cbDataWrite function**
 When the application program prepares and stores outgoing data intended for the PROFINET master in the output variables (PN_sendContainer) of the stack interface, the PN_APP_IOD_cyclicDataTask invokes this function to transfer the outgoing data from the output variables (PN_sendContainer) to the PRU-ICSS. The PRU-ICSS then transmits the data to the PROFINET master, ensuring seamless real-time communication.

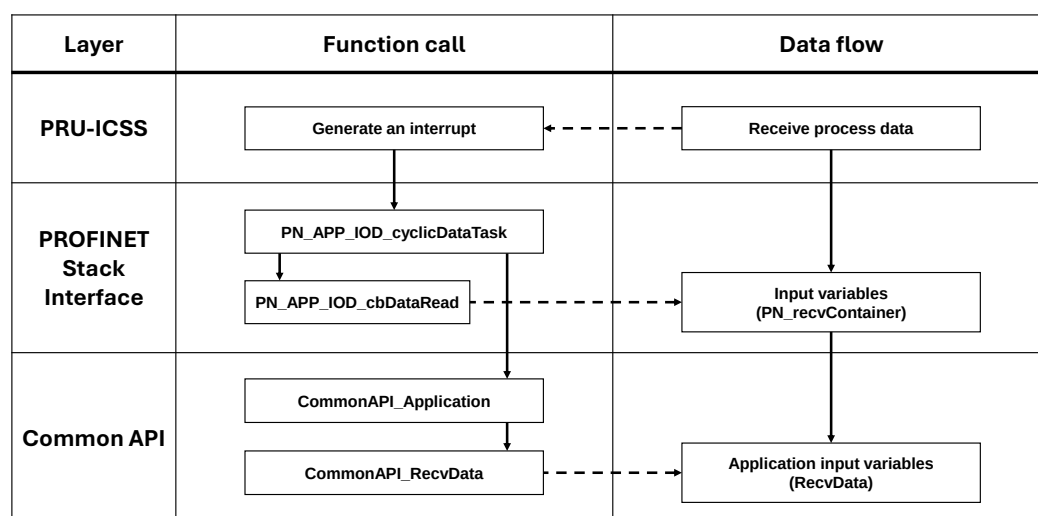


Figure 5. Function call procedure and data flow during the incoming data reception in the PROFINET protocol.

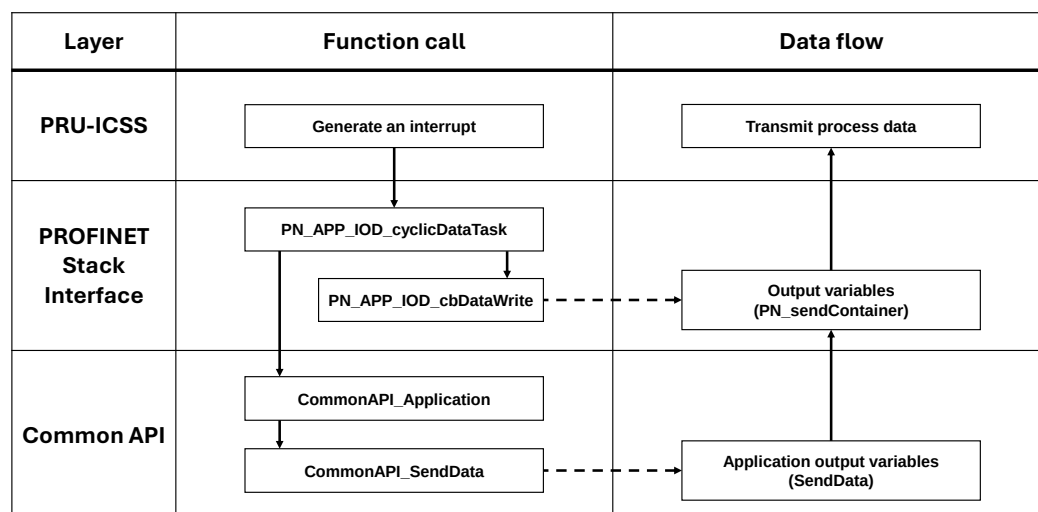


Figure 6. Function call procedure and data flow during the outgoing data transmission in the PROFINET protocol.

3.4.2. Common API Layer for PROFINET

For the PROFINET common API layer, several functions are implemented to provide a high-level interface for managing communication and operations within the PROFINET slave device. These functions facilitate seamless application program development using the PROFINET protocol stack. The developed API functions simplify process data exchange and task execution, ensuring efficient and reliable real-time communication. Notably, these functions are designed to be commonly utilized for the EtherCAT and EtherNet/IP protocols as well. In the following, we detail the roles and functionalities of the developed functions for the PROFINET common API layer, emphasizing their design and implementation for seamless PROFINET protocol integration.

- **CommonAPI_Init function**
This function initializes the hardware and protocols of the slave device while also creating the necessary FreeRTOS task.
- **CommonAPI_Application function**
This function acts as a wrapper, invoked by the PN_APP_IOD_cyclicDataTask of the PROFINET stack interface layer to execute the slave application program in the common API layer, as illustrated in Figures 5 and 6.
- **CommonAPI_RecvData**
This function is designed to handle the reception of incoming data on a per-byte basis. It retrieves data from the input variables (PN_recvContainer) of the PROFINET stack interface, which have stored the incoming data transferred from the PRU-ICSS via the PN_APP_IOD_cbDataRead function. The remaining functionality and description of this function are identical to those of the commonAPI_RecvData function in the common API for the EtherCAT protocol. Figure 5 illustrates the data flow when this function is executed.
- **CommonAPI_SendData**
This function is designed to handle the transmission of outgoing data on a per-byte basis. It writes the outgoing data value to the output variables (PN_sendContainer) of the PROFINET stack interface, which store the outgoing data to be transferred to the PRU-ICSS via the PN_APP_IOD_cbDataWrite function. The remaining functionality and description of this function are identical to those of the CommonAPI_RecvData function in the common API for the EtherCAT protocol. Figure 6 illustrates the data flow when this function is executed.

Table 2 provides a list of all functions and FreeRTOS tasks developed for the common API and PROFINET stack interface to enable efficient data communication in the PROFINET protocol.

Table 2. Functions and FreeRTOS task developed for the common API and stack interface of the PROFINET protocol.

Layer	Function	FreeRTOS Task
PROFINET stack interface	PN_APP_IOD_cbDataRead	PN_APP_IOD_cyclicDataTask
	PN_APP_IOD_cbDataWrite	
Common API	CommonAPI_Init	
	CommonAPI_Application	
	CommonAPI_RecvData	
	CommonAPI_SendData	

3.5. EtherNet/IP Integration

Similar to the EtherCAT and PROFINET protocols, the EtherNet/IP physical and link layers are implemented using TI's PRU-ICSS along with its PRU firmware. The EtherNet/IP slave stack is obtained from the TI's Industrial Communications SDK for AM243x (09.02.00.15). To enable seamless operation of the EtherNet/IP protocol through a unified API, it is necessary to design and implement both a suitable EtherNet/IP stack interface and a common API. The EtherNet/IP stack interface plays a critical role in facilitating real-time data exchange, which is achieved by developing a callback function synchronized with incoming data reception. This synchronization is effectively managed using the FreeRTOS synchronization mechanism. In the following, we detail the design and implementation of the stack interface layer and the common API layer for EtherNet/IP integration.

3.5.1. EtherNet/IP Stack Interface Layer

For the EtherNet/IP stack interface, a callback function is implemented to manage process data communication within an EtherNet/IP slave device. The EtherNet/IP stack interface processes incoming data from the EtherNet/IP master by storing it from the PRU-ICSS to the input variables of the stack interface. Conversely, this interface transfers outgoing data, which are prepared and stored in the output variables of the stack interface by the application program, back to the PRU-ICSS. Subsequently, the EtherNet/IP link and physical layer implemented on PRU-ICSS transmits the outgoing data to the EtherNet/IP master. In the following, we detail the roles and functionalities of the developed callback function for the EtherNet/IP stack interface layer, emphasizing their design and implementation for seamless EtherNet/IP protocol integration.

- **EI_APP_GENERIC_DEVICE_run** function
This callback function in the EtherNet/IP stack interface is invoked by a PRU-ICSS interrupt when the PRU-ICSS receives incoming process data from the master. Upon activation, the function stores the incoming data from the PRU-ICSS into the input variables (EIP_recvContainer) defined in the stack interface layer. Following this, the function executes the application program in the common API layer, where the incoming data are retrieved and processed. The application program then prepares the outgoing data and stores them in the output variable (EIP_sendContainer), also defined in the stack interface layer. Subsequently, the function transfers the outgoing data to the PRU-ICSS, which transmits them to the EtherNet/IP master, ensuring seamless real-time communication. The overall function call sequence and the data flow

managed by the EI_APP_GENERIC_DEVICE_run callback function are illustrated in Figures 7 and 8.

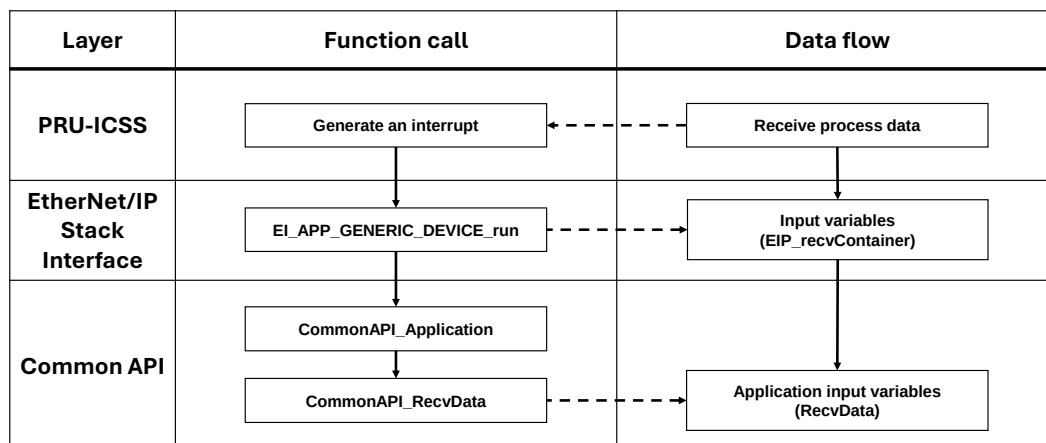


Figure 7. Function call procedure and data flow during the incoming data reception in the EtherNet/IP protocol.

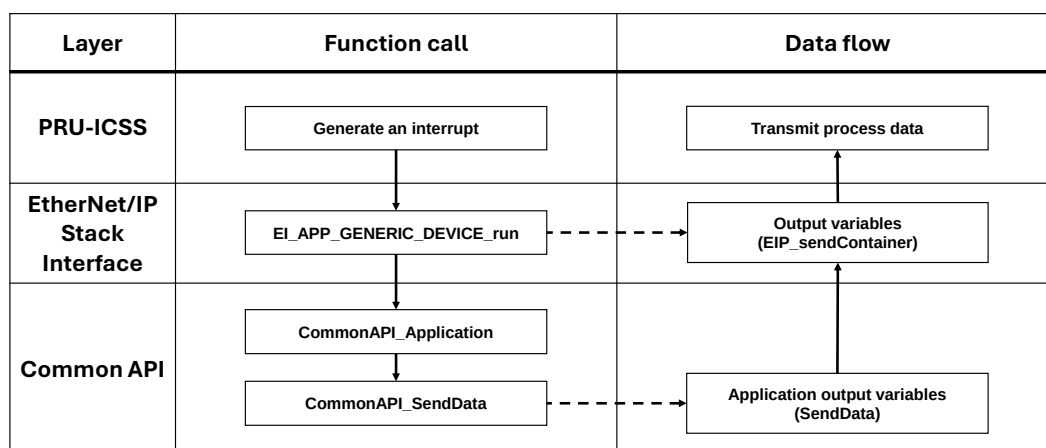


Figure 8. Function call procedure and data flow during the outgoing data transmission in the EtherNet/IP protocol.

3.5.2. Common API Layer for EtherNet/IP

For the EtherNet/IP common API layer, several functions are implemented to provide a high-level interface for managing communication and operations within the EtherNet/IP slave device. These functions facilitate seamless application program development using the EtherNet/IP protocol stack. The developed API functions simplify process data exchange and task execution, ensuring efficient and reliable real-time communication. Notably, these functions are designed to be commonly utilized for the EtherCAT and PROFINET protocols as well. In the following, we detail the roles and functionalities of the developed functions for the EtherNet/IP common API layer, emphasizing their design and implementation for seamless EtherNet/IP protocol integration.

- **CommonAPI_Init function**
This function initializes the hardware and protocols of the slave device.
- **CommonAPI_Application function**
This function acts as a wrapper, invoked by the EI_APP_GENERIC_DEVICE_run callback function of the EtherNet/IP stack interface layer to execute the slave application program in the common API layer, as illustrated in Figures 7 and 8.

- **CommonAPI_RecvData function**
This function is designed to handle the reception of incoming data on a per-byte basis. It retrieves data from the input variables (EIP_recvContainer) of the EtherNet/IP stack interface, which have stored the incoming data transferred from the PRU-ICSS via the EI_APP_GENERIC_DEVICE_run callback function. The remaining functionality and description of this function are identical to those of the CommonAPI_RecvData function in the common API for the EtherCAT protocol. Figure 7 illustrates the data flow when this function is executed.
- **CommonAPI_SendData function**
This function is designed to handle the transmission of outgoing data on a per-byte basis. It writes the outgoing data value to the output variables (EIP_sendContainer) of the EtherNet/IP stack interface, which store the outgoing data to be transferred to the PRU-ICSS via the EI_APP_GENERIC_DEVICE_run callback function. The remaining functionality and description of this function are identical to those of the CommonAPI_RecvData function in the common API for the EtherCAT protocol. Figure 8 illustrates the data flow when this function is executed.

Table 3 provides a list of all functions and FreeRTOS tasks developed for the common API and EtherNet/IP stack interface to enable efficient data communication in the EtherNet/IP protocol.

Table 3. Callback function developed for the common API and stack interface of the EtherNet/IP protocol.

Layer	Function
EtherNet/IP stack interface	EI_APP_GENERIC_DEVICE_run
Common API	CommonAPI_Init
	CommonAPI_Application
	CommonAPI_RecvData
	CommonAPI_SendData

4. Experiments and Results

4.1. Setup

We conduct a series of experiments to evaluate the functionality and reliability of the developed common API and stack interfaces for three Ethernet-based fieldbus protocols: EtherCAT, PROFINET, and EtherNet/IP. The experimental setup for the fieldbus network is built using Beckhoff's TwinCAT 3.1, running on a Windows PC as the master and a TI TMD243EVM board based on the AM2434 MCU as the slave device. The detailed specifications of the Windows PC used for the master are as follows: Intel Core i9-10900 CPU, 32 GB RAM, Intel UHD Graphics 630, Intel Ethernet Connection I219-V LAN card, and Windows 10 Pro 64-bit operating system. The configuration of the setup is illustrated in Figure 9.

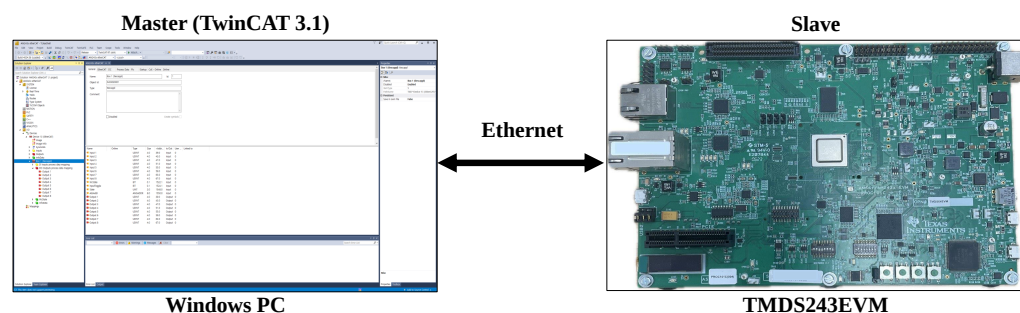


Figure 9. Experimental setup for testing the common API and stack interface.

The exchange data structure is configured to handle a total of 64 bytes of protocol process data, consisting of 32 bytes for transmission and 32 bytes for reception. The 32 bytes of transmission data, generated by the master and sent to the slave, are divided into eight 4-byte segments labeled as Output1 through Output8. Similarly, the 32 bytes of reception data, generated by the slave and sent to the master, are divided into eight 4-byte segments labeled as Input1 through Input8.

To implement this process data structure for each protocol, it is necessary to generate specific configuration files, including an ESI file for EtherCAT, a GSD file for PROFINET, and an EDS file for EtherNet/IP. These files encapsulate the process data structure information, as illustrated in Figures 10–13.

The experimental scenario to validate the developed common API and stack interfaces is as follows: The master periodically sends eight predetermined 4-byte segments (32 bytes total) through the transmission process data, Output1 to Output8. The slave receives this 32-byte dataset, processes it, and then returns the same eight 4-byte segments to the master via the reception process data, Input1 to Input8. Additionally, upon receiving the 32-byte process data from the master, the slave controls four LED devices on the TMDS243EVM board to represent the binary state of the first byte of the first received process data segment, Output1, by turning the LEDs on or off accordingly.

Direction	Index	Name	SubIndex	Subname	Default Value
Output (Master to slave)	0x1C12	RxPDO assign	-	-	0x1601
	0x1601	Digital Output process data mapping	01	SubIndex001	0x7010 01 20
			02	SubIndex002	0x7010 02 20
			03	SubIndex003	0x7010 03 20
			04	SubIndex004	0x7010 04 20
			05	SubIndex005	0x7010 05 20
			06	SubIndex006	0x7010 06 20
			07	SubIndex007	0x7010 07 20
			08	SubIndex008	0x7010 08 20
	0x7010	Digital Output	01	Output1	0
			02	Output2	0
			03	Output3	0
			04	Output4	0
			05	Output5	0
			06	Output6	0
			07	Output7	0
			08	Output8	0

Figure 10. Configuration of EtherCAT output process data mapping in the ESI file.

Direction	Index	Name	SubIndex	Subname	Default Value
Input (Slave to master)	0x1C13	TxPDO assign	-	-	0x1A00
	0x1A00	Digital Input process data mapping	01	SubIndex001	0x6000 01 20
			02	SubIndex002	0x6000 02 20
			03	SubIndex003	0x6000 03 20
			04	SubIndex004	0x6000 04 20
			05	SubIndex005	0x6000 05 20
			06	SubIndex006	0x6000 06 20
			07	SubIndex007	0x6000 07 20
			08	SubIndex008	0x6000 08 20
	0x6000	Digital Input	01	Input1	0
			02	Input2	0
			03	Input3	0
			04	Input4	0
			05	Input5	0
			06	Input6	0
			07	Input7	0
			08	Input8	0

Figure 11. Configuration of EtherCAT input process data mapping in the ESI file.

Slot	Module	Subslot	Submodule Name	Detail
0	PN_CFG_IDD_DAP_MOD_ID (Device Access Point (DAP) Module)	1	PN_CFG_IDD_DAP_SUB_ID	DAP submodule
		0x8000	PN_CFG_IDD_DAP_IF_ID	PDEV (Physical Device) interface
		0x8001	PN_CFG_IDD_DAP_P1_ID	PDEV port1
		0x8002	PN_CFG_IDD_DAP_P2_ID	PDEV port2
1	PN_CFG_IDM_MOD_IO_ID (I/O Data Module)	1	PN_CFG_IDS_IO_IF_ID	IO subslot interface
		2	PN_CFG_IDS_IO_T4_ID	4-byte input submodule and 4-byte output submodule for each subslot (Input1 to Input8, Output1 to Output8)
		3		
		4		
		5		
		6		
		7		
		8		
		9		

Figure 12. Configuration of PROFINET process data mapping in the GSD file.

Class ID	Instance ID	Attribute ID	Attribute Name	Detail
0x70	0x01	0x0368	Output1	Output 4-byte attribute (Master to Slave)
		0x0369	Output2	
		0x036a	Output3	
		0x036b	Output4	
		0x036c	Output5	
		0x036d	Output6	
		0x036e	Output7	
		0x036f	Output8	
		0x0360	Input1	Input 4-byte attribute (Slave to Master)
		0x0361	Input2	
		0x0362	Input3	
		0x0363	Input4	
		0x0364	Input5	
		0x0365	Input6	
		0x0366	Input7	
		0x0367	Input8	

Figure 13. Configuration of EtherNet/IP process data mapping in the EDS file.

To realize this experimental scenario, a common application program is developed and used commonly across all three protocols. The common application program is structured as a function, as illustrated in Figure 14, and is called by the stack interface whenever process data are received from the master.

Application Program	Description
<pre>void CommonAPI_Application() { uint8_t recv = 0; int i; recv = CommonAPI_RecvData(1); LedControl(recv); CommonAPI_SendData(recv,1); for(i=2; i<33; i++) { recv = CommonAPI_RecvData(i); CommonAPI_SendData(recv,i); } }</pre>	- Variables are declared and initialized. - The first byte of the received process data is returned and stored in the recv variable. - The LED light on the board is controlled such that the first byte of the received process data is displayed in the binary number. - The first byte of the received process data is written to the first byte of the output variable (sendcontainer) of each stack interface. - The other bytes of the received process data are returned and stored in the recv variable. - The i-th byte of the received process data is returned and stored in the recv variable. - The i-th byte of the received process data is written to the i-th byte of the output variable (sendcontainer) of each stack interface

Figure 14. Application program developed for implementing the experimental scenario.

4.2. Result

Figures 15 and 16a present the experimental results for the EtherCAT protocol. As illustrated in Figure 15, the EtherCAT master transmits sequential data to the EtherCAT slave device, incrementing from 01 to 32 in each byte of the process data, labeled Output1 to Output8. The slave device receives these process data and returns each value to the master via the corresponding process data, Input1 to Input8. A comparison of the input and output process data on the master, as illustrated in Figure 15, confirms that the input and output data are identical, thereby verifying that the developed common API and stack interface for EtherCAT operates correctly without error. Additionally, the first byte of Output1 is 01, and Figure 16a demonstrates that the LED device on the TMDS243EVM displays the binary representation of 01 as 0001₂. This LED display further confirms the accurate operation of the developed API with no observed errors.

Similarly, Figures 17 and 16b present the experimental results for the PROFINET protocol. As illustrated in Figure 17, the PROFINET master transmits sequential data to the PROFINET slave device, incrementing from 02 to 33 in each byte of the process data, labeled Output1 to Output8. The slave device receives these process data and returns each value to the master via the corresponding process data, Input1 to Input8. A comparison of the input and output process data on the master, as illustrated in Figure 17, confirms that the input and output data are identical, thereby verifying that the developed common API and stack interface for PROFINET operates correctly without error. Additionally, the first byte of Output1 is 02, and Figure 16b demonstrates that the LED device on the TMDS243EVM displays the binary representation of 02 as 0010₂. This LED display further confirms the accurate operation of the developed common API and stack interface with no observed errors.

AM243x EtherCAT		AM243x EtherCAT	
Name	Online	Name	Online
Output 1	0x01020304	Input 1	0x01020304
Output 2	0x05060708	Input 2	0x05060708
Output 3	0x09101112	Input 3	0x09101112
Output 4	0x13141516	Input 4	0x13141516
Output 5	0x17181920	Input 5	0x17181920
Output 6	0x21222324	Input 6	0x21222324
Output 7	0x25262728	Input 7	0x25262728
Output 8	0x29303132	Input 8	0x29303132

Figure 15. Data transmitted and received by TwinCAT during the EtherCAT experiment.

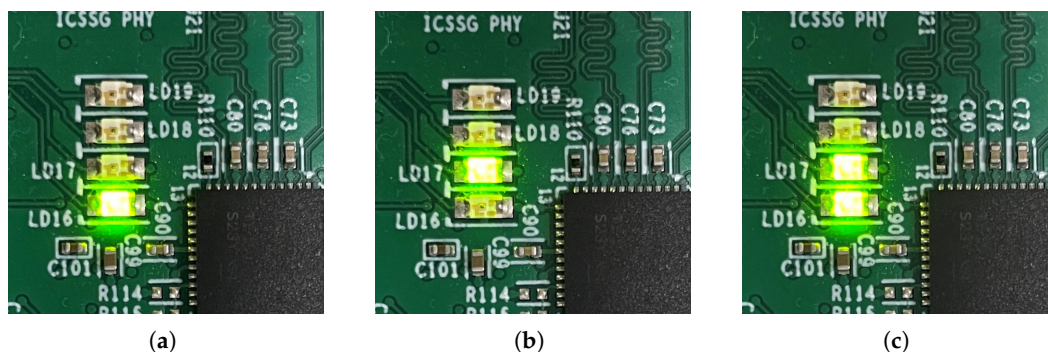


Figure 16. LED status of TMD5243EVM board. (a) During EtherCAT experiment. (b) During PROFINET experiment. (c) During EtherNet/IP experiment.

Similarly, Figures 18 and 16c present the experimental results for the EtherNet/IP protocol. As illustrated in Figure 18, the EtherNet/IP master transmits sequential data to the EtherNet/IP slave device, incrementing from 03 to 34 in each byte of the process data, labeled Output1 to Output8. The slave device receives these process data and returns each value to the master via the corresponding process data, Input1 to Input8. A comparison of the input and output process data on the master, as illustrated in Figure 18, confirms that the input and output data are identical, thereby verifying that the developed common API and stack interface for EtherNet/IP operates correctly without error. Additionally, the first byte of Output1 is 03, and Figure 16c demonstrates that the LED device on the TMD5243EVM displays the binary representation of 03 as 0011₂. This LED display further confirms the accurate operation of the developed common API and stack interface with no observed errors.

AM243x PROFINET		AM243x PROFINET	
Name	Online	Name	Online
Output 4-Byte	0x02030405	Input 4-Byte	0x02030405
Output 4-Byte	0x06070809	Input 4-Byte	0x06070809
Output 4-Byte	0x10111213	Input 4-Byte	0x10111213
Output 4-Byte	0x14151617	Input 4-Byte	0x14151617
Output 4-Byte	0x18192021	Input 4-Byte	0x18192021
Output 4-Byte	0x22232425	Input 4-Byte	0x22232425
Output 4-Byte	0x26272829	Input 4-Byte	0x26272829
Output 4-Byte	0x30313233	Input 4-Byte	0x30313233

Figure 17. Data transmitted and received by TwinCAT during the PROFINET experiment.

AM243x EtherNet_IP		AM243x EtherNet_IP	
Name	Online	Name	Online
Output1	0x03040506	Input1	0x03040506
Output2	0x07080910	Input2	0x07080910
Output3	0x11121314	Input3	0x11121314
Output4	0x15161718	Input4	0x15161718
Output5	0x19202122	Input5	0x19202122
Output6	0x23242526	Input6	0x23242526
Output7	0x27282930	Input7	0x27282930
Output8	0x31323334	Input8	0x31323334

Figure 18. Data transmitted and received by TwinCAT during the EtherNet/IP experiment.

5. Conclusions

In this study, we present a novel solution to address the challenges of protocol switching in Ethernet-based fieldbus networks by developing a common API and stack interfaces.

The proposed solution eliminates the need for extensive reprogramming and replacement of protocol stacks and Ethernet drivers when switching the Ethernet-based fieldbus protocols among EtherCAT, PROFINET, and EtherNet/IP. By leveraging the synchronization mechanism of FreeRTOS, the stack interfaces enable seamless real-time data exchange between the API and the respective protocol stacks.

The developed common API and stack interfaces not only simplify the development of universally compatible slave device applications but also automate the selection and configuration of protocol stacks and Ethernet drivers, further reducing development complexity and effort. The implementation on TI's TMDS243EVM board and validation through a fieldbus network demonstrate the solution's functionality, reliability, and practical applicability.

These findings highlight the potential of the proposed common API and stack interfaces to streamline protocol management in Ethernet-based fieldbus networks, significantly enhancing system flexibility, development efficiency, and cost-effectiveness. The developed common API currently supports only three protocols—EtherCAT, PROFINET, and EtherNet/IP—on AM2434 MCU-based embedded platforms. However, future research could expand this approach to support additional protocols (e.g., POWERLINK, Modbus TCP, and IO-Link) and MCUs, as well as further optimize performance across diverse hardware platforms.

Author Contributions: Conceptualization and methodology, D.K.; software, D.K.; validation, D.K.; formal analysis, D.K.; investigation, J.-Y.C.; resources, D.K.; data curation, D.K.; writing—original draft preparation, D.K.; writing—review and editing, D.K. and J.-Y.C.; visualization, D.K.; supervision, J.-Y.C.; project administration, J.-Y.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by DN solutions, Korea.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Oztemel, E.; Gursev, S. Literature review of Industry 4.0 and related technologies. *J. Intell. Manuf.* **2020**, *31*, 127–182. [[CrossRef](#)]
2. Mulongo, N.Y.; Lusamba, M. Evolution of Industry 4.0 In Manufacturing Sectors: Insight On Prisma Approach. In Proceedings of the 2024 International Symposium on Networks, Computers and Communications, Washington, DC, USA, 22–25 October 2024.
3. Shaikh, M.; Shah, P.; Sekhar, R. Communication Protocols in Industry 4.0. In Proceedings of the 2023 International Conference on Sustainable Emerging Innovations in Engineering and Technology, Ghaziabad, India, 14–15 September 2023.
4. Pasupuleti, M.K. *Transformative Innovations in Smart Manufacturing*, 1st ed.; IGI Global: Hershey, PA, USA, 2024; pp. 119–133.

5. Calderón, D.; Folgado, F.J.; González, I.; Calderón, A.J. Implementation and Experimental Application of Industrial IoT Architecture Using Automation and IoT Hardware/Software. *Sensors* **2024**, *24*, 8074. [\[CrossRef\]](#)
6. Behnke, I.; Austad, H. Real-Time Performance of Industrial IoT Communication Technologies: A Review. *IEEE Internet Things J.* **2024**, *11*, 7399–7410. [\[CrossRef\]](#)
7. Petkov, N.; Naumov, A. Overview of industrial communication in process automation. In Proceedings of the 2022 International Conference Automatics and Informatics, Varna, Bulgaria, 6–8 October 2022.
8. Wu, X.; Xie, L. Performance evaluation of industrial Ethernet protocols for networked control application. *Control. Eng. Pract.* **2019**, *84*, 208–217. [\[CrossRef\]](#)
9. Vitturi, S.; Sauter, T.; Pang, Z. Real-time networks and protocols for factory automation and process control systems. *Proc. IEEE* **2019**, *107*, 939–943. [\[CrossRef\]](#)
10. Danielis, P.; Skodzik, J.; Altmann, V.; Schweissguth, E.B.; Golasowski, F.; Timmermann, D.; Schacht, J. Survey on real-time communication via ethernet in industrial automation environments. In Proceedings of the Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, Barcelona, Spain, 12 January 2015.
11. MacPherson, J.; Dawson, M. Common protocol to support disparate communication types within industrial Ethernet environments. *Int. J. Netw. Sci.* **2016**, *1*, 134–154. [\[CrossRef\]](#)
12. Koyasako, Y.; Suzuki, T.; Hatano, T.; Shimada, T.; Yoshida, T. Demonstration of Industrial Ethernet Protocol Softwarization and Advanced Motion Control for Full Software-Defined Factory Network. *IEEE Access* **2024**, *12*, 104020–104030. [\[CrossRef\]](#)
13. Han, D.; Zhao, Y.; Gong, Y. Design of Multi-Protocol Industrial Ethernet Security Monitor. In Proceedings of the 2021 IEEE International Conference on Power Electronics, Computer Applications, Shenyang, China, 22–24 January 2021.
14. Berlemann, L.; Pabst, R.; Schinnenburg, M.; Walke, B. Reconfigurable multi-mode protocol reference model facilitating modes convergence. In Proceedings of the 11th European Wireless Conference 2005-Next Generation wireless and Mobile Communications and Services, Nicosia, Cyprus, 10–13 April 2005.
15. Spale, J. netX-Network controller for automation. In Proceedings of the 2009 Applied Electronics, Pilsen, Czech Republic, 9–10 September 2009.
16. Dzung, D.; Naedele, M.; Von Hoff, T.; Crevatin, M. Security for Industrial Communication Systems. *Proc. IEEE* **2005**, *93*, 1152–1177. [\[CrossRef\]](#)
17. Jansen, D.; Buttner, H. Real-time Ethernet: The EtherCAT solution. *Comput. Control. Eng.* **2004**, *15*, 16–21. [\[CrossRef\]](#)
18. Nguyen, V.Q.; Jeon, J.W. EtherCAT network latency analysis. In Proceedings of the 2016 International Conference on Computing, Communication and Automation, Greater Noida, India, 29–30 April 2016.
19. Feld, J. PROFINET-scalable factory communication for all applications. In Proceedings of the IEEE International Workshop on Factory Communication Systems, Vienna, Austria, 22–24 September 2004.
20. Yang, M.; Li, G. Analysis of PROFINET IO communication protocol. In Proceedings of the 2014 Fourth International Conference on Instrumentation and Measurement, Computer, Communication and Control, Harbin, China, 29 December 2014.
21. Warren, J.C. Ethernet/IP applications for electrical industrial systems. In Proceedings of the 2009 IEEE Industry Applications Society Annual Meeting, Houston, TX, USA, 4–8 October 2009.
22. Brooks, P. Ethernet/IP-industrial protocol. In Proceedings of the ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation, Proceedings, Antibes-Juan les Pins, France, 15–18 October 2001.
23. Reeder, J.; Chitnis, K.; Hannah, M.; Mody, M.; Yennacheti, R.V.; Sivasankaran, S.; Livingston, F.; Khandla, D.; Govindarajan, S.; Sivaraj, R.; et al. Single Chip Connected Multi-Axis Servo Drive for Industrial Systems. In Proceedings of the 2022 IEEE International Conference on Consumer Electronics, Las Vegas, NV, USA, 7–9 January 2022.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.