Ausarbeitung eines Konzepts für Ergonomische Arbeitsplätze einer Montagelinie

Studienarbeit im Rahmen des AWP
"Problemlösungen in der Praxis"

Vorgelegt von:

Felix Dick
Matrikelnummer: 22111369
Maximilian Duschl
Matrikelnummer: 00756375

Betreuer:
NAME EINFÜGEN————————-

Deggendorf, den June 23, 2025

Contents

1 Introduction

This report summarizes the results of the laboratory exercises conducted in the lecture "AI-M-1: Selected Topics of Embedded Software Development I - Embedded Connectivity (SS25)". This includes general tasks, as well as group-specific tasks. They were worked on in the following order:

At the beginning a preparation task was completed, which helps to familiarize the user with the setup and lets him collect important data for the following tasks. The first practical tasks involved establishing basic point-to-point communication using the ANDi tool in loopback mode. The knowledge and scripts developed in the loopback setup were then transferred to the real-world scenario, connecting devices within the lab to observe behaviour and performance under realistic conditions. Finally, a connection is established between the gateway of the user's own workstation and the lab's central Media Gateway to access the camera connected to it.

Upon completion of these core assignments, the group of this report worked on the topic "Automotive MicroController - Aurix ". The goal was to realize a CAN Ethernet Gateway with an "Infineon TriCore Board".

2 Preparation Part: Workstation Network Structure Analysis

Before initiating synchronization procedures, an extensive analysis of the local workstation network was required. The analysis aimed at revealing all devices within the 192.168.0.0/24 subnet, collecting relevant IP and MAC addresses essential for the ensuing Precision Time Protocol (PTP) configuration.

The first step was to use **ifconfig**, which revealed the following configuration details:

- Host IP: 192.168.0.227
- Host MAC: 00-13-3B-B0-10-B5

Additionally, Oracle VM VirtualBox hosted a Linux virtual machine, which was verified for proper connectivity within the network:

- VM IP: 192.168.0.28
- VM MAC: 08-00-27-8D-C8-A2

Zenmap was used to run the network scan with nmap. Conducting Ping and Intense scans within the subnet revealed several accessible devices.

- 192.168.0.173 (ports 88, 443, 888; Webcam)
- 192.168.0.226 (ports 1309, 3389; Workstation)
- 192.168.0.49 (port 80; Media Gateway)

3 Part 1: ANDi Tool and Loopback

This chapter introduces the initial setup and basic testing procedures.

3.1 Introduction to the ANDI-tool

ANDi (Automotive Network Diagnoser) is a cross-platform test and analysis tool for automotive electronic networks, designed to support software and ECU development at every stage. Its core functions are to simulate network traffic, execute component tests and analyse the resulting data[1]. It supports simulation, monitoring, and analysis of various network protocols, including CAN, LIN, FlexRay, and Ethernet (BroadR-Reach).

3.2 Introduction to Wireshark

Wireshark is a widely used open-source network protocol analyser. Its primary purpose is to capture and inspect data packets travelling across a network, enabling detailed analysis of network traffic. Key features include real-time packet capture, deep inspection of hundreds of protocols, filtering capabilities, and the ability to reconstruct data streams.

3.3 Loopback Test Procedure

- Step 1: Connect the hardware: Attach the ANDi device to the CAN bus or network interface intended for testing, ensuring all physical connections are secure and correct.
- Step 2: Configure the software: Launch the ANDi software, set up the interface parameters (bitrate, channel, etc.), and enable loopback mode in the configuration settings.
- Step 3: Run the test and observe results: Start the loopback test, send messages through the interface, and monitor received frames to verify correct transmission and reception within the same device.

3.4 Task Report

The task utilized the ANDi tool in combination with Wireshark to establish communication through loopback testing. To achieve this, a stimulation and a logging adapter had to be selected. The stimulation adapter should transmit frames or signals, while the logging adapter is used to receive incoming packets.

After running a loopback script included in the task the packages were captured as expected, but only after selecting the right adapter for both (Figure 1). The layer 2 frame was received as expected (Figure 2) and could be traced in Wireshark.

The next step was to create 2 distinct scripts, to distinguish between sending and receiving frames. Additionally the frames should be more distinguishable from another, so the requirement was to give each frame an identifier corresponding to the order they were sent in with 20 frames being sent over the course of 20 seconds.

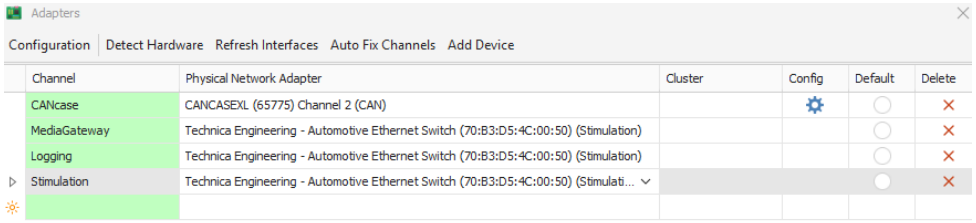

Figure 1: Selection of the same adapter for Logging and Stimulation.

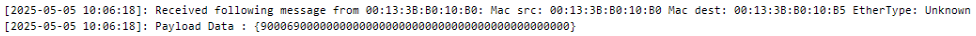

Figure 2: Received frame from loopback.

After achieving this, subsequent task were to test further ANDi tools like SendPing and Burst Sending and analyze the traffic: Send Ping sends one request and awaits a reply while Burst Sending sends multiple requests in a short time without waiting for a reply first. Both operate on Layer 3 and use Ethernet frames on Layer 2 with the key difference being the number and timing of packets sent.

After changing the communication from Layer 2 (MAC address) to a layer 3 (IP address) connection, the data frames change. They reveal more information about the connection, TTL, ports, and the checksum of the packages.

3.5 Conclusion

In this package of Tasks basic understanding of packet construction and data traffic was developed by experimenting with a simulated point to point connection. The tasks were fairly basic and deepened the theoretical understanding of the topic, ensuring fundamental knowledge needed for subsequent tasks. No difficult challenges were encountered and the workflow was straightforward.

4 Part 2: ANDi Tool and Automotive Ethernet

This section details the establishment of a communication path.

4.1 Simple Connection Setup

A simple setup typically means connecting two or more devices through the gateway, which acts as an intermediary for data transmission. For that devices are physically linked to the MediaGateway's network ports using twisted pair cables and MediaConverters. The MediaGateway acts as a Layer 2 switch in this setup, forwarding Ethernet frames between connected devices without complex routing. Each device connects via its Ethernet port to a MediaConverter, which then connects to the MediaGateway. The gateway ports are set to slave mode by default, while the MediaConverters are configured as master. Logically, this setup establishes a direct Layer 2 link between the devices, allowing them to exchange Ethernet frames transparently through the MediaGateway.

4.2 VLAN Configuration

VLANs (Virtual Local Area Networks) allow segmenting a physical network into multiple logical networks. In the context of the laboratory setup, VLANs create isolated broadcast domains within the MediaGateway, ensuring that only devices with matching VLAN tags can communicate with each other. Frames are tagged with a VLAN ID, and only those with a matching ID are forwarded within that VLAN. This improves bandwidth utilization and enhances security by isolating traffic, as frames without the correct VLAN tag cannot enter the VLAN and by thatunwanted access or interference is prevented.

To configure VLANs on the MediaGateway, you first assign VLAN IDs to internal ports and mark them as either tagged or untagged, depending on the connected device's support. Since Windows does not support VLAN tagging natively, ports connecting to Windows devices are usually set as untagged members of a VLAN. Configuration is done via the MediaGateway's web interface, where you enter the default IP address, enable IEEE 802.1Q VLAN mode, assign VLAN IDs to relevant ports, and then save and restart the gateway to apply the settings. After proper VLAN configuration, devices connected to the MediaGateway can communicate securely and efficiently within their assigned VLANs.

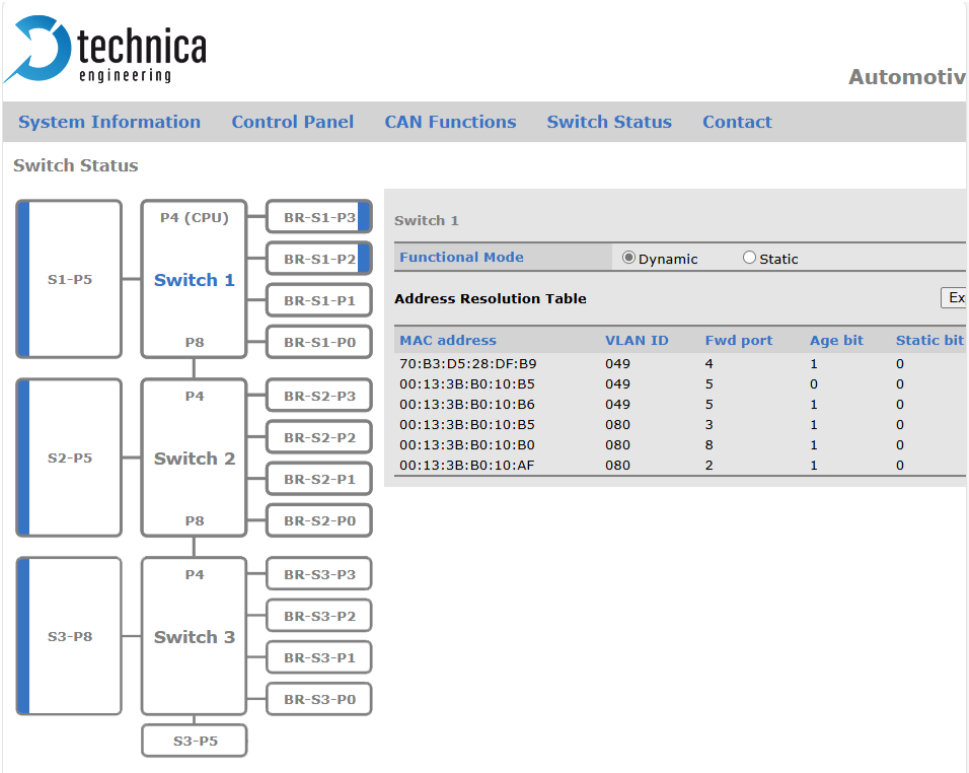

Figure 3: VLAN Configuration.

4.3 Task Report

After realizing a simulated point to point connection in Part 1, this set of tasks revolved around advancing towards direct network communication through the Media Gateway

with another physical workstation. The unconfigured media gateway will act as a layer 2 switch. Using the scripts from Part 1 as basis, they as well as the adapters had to be updated to reflect the new target.

The next task instructed the user to configure VLAN and build up a connection using it. Subsequently, a VLAN-based transmission path was configured, necessitating adjustments in the Media Gateway port settings (Figure 3). Upon correct VLAN implementation, packets containing payload data with increasing values were successfully transmitted and logged in Wireshark. The last task of this set, to access the webcam, was successful as well.

Significant challenges in this part included selecting the correct adapters and finding the correct port configuration, which was confusing at first, but after understanding the logic behind it, presented itself as a trivial matter. Another challenge was troubleshooting physical connectivity issues with an unresponsive webcam, which was later revealed to be the fault of a missing physical connection to the gateway.

4.4 Conclusion

In this package of Tasks the point to point connection to another workstation using the MediaGateway with the MAC address at first, then via VLAN-based transmission, was configured and established. The tasks helped further the understanding of VLAN and network traffic. There were some challenges in solving the tasks, but they were manageable to work around.

5 Part 3: ANDi Tool and Automotive Ethernet in the lab

This chapter covers the integration of the setup into a larger network.

5.1 Network Architecture Overview

In Part 3 of the workshop, the target network architecture for integrating the setup consists of multiple workstations connected via individual MediaGateways to a central MediaGateway. Additionally, a webcam is connected to the central Gateway. Each workstation's Gateway links through specific ports to the central gateway, creating a star-like topology. This central Gateway manages traffic between the different workstations and shared devices such as the central webcam, enabling communication beyond simple point-to-point connections. This architecture builds on the baseline established in earlier parts, where basic communication paths and VLAN configurations were performed between two devices over a single MediaGateway.

5.2 Task Report

The assigned tasks consisted of establishing a functional communication path between the workstation's MediaGateway and the central MediaGateway, with the objective of accessing the central webcam. The process involved configuring each MediaGateway's

ports with the correct VLAN IDs and ensuring the physical connections to the central MediaGateway are correctly set up.

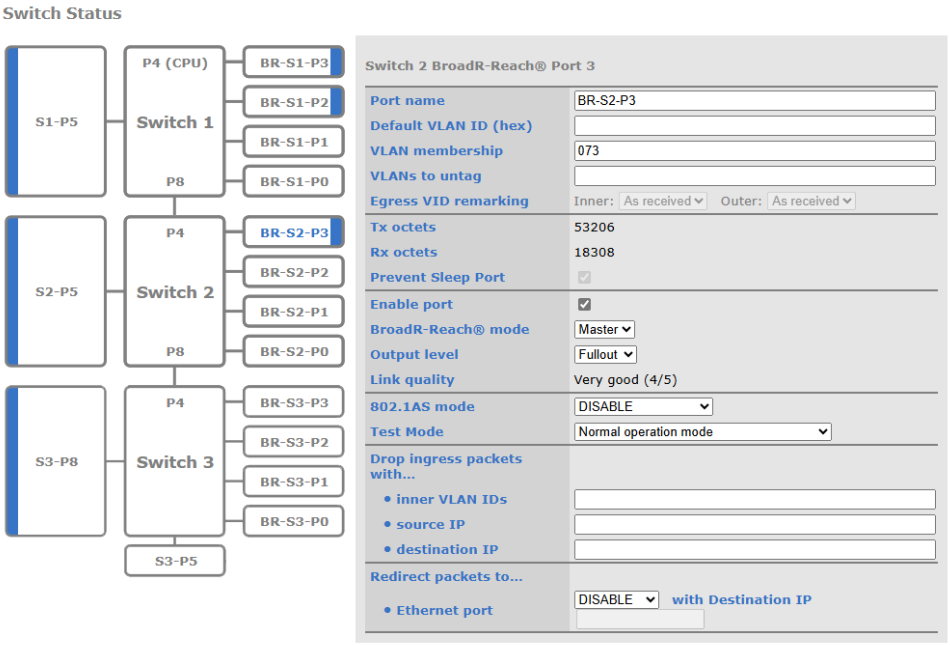

Figure 4: VLAN Configuration.

After setting up the configuration (Figure 4) a connection to the webcam could not be established, despite everything seemingly being correctly set up. The webcam stayed unresponsive to pings and even the instructor was unsure what was wrong. A restart of the central gateway and the central webcam was issued to troubleshoot, which solved the problem, despite the reason for the error staying unidentified. Post-reboot, connectivity was confirmed via successful ping responses. Remote access to the central webcam could be successfully established through a web browser . Concurrently, network traffic was captured using Wireshark, which verified the presence of correctly tagged Ethernet frames and validated the proper routing of data through the VLANs.

5.3 Conclusion

In this package of Tasks the understanding of working with VLAN was deepened. The proficiency to solving the tasks profited from the gained understanding from prior parts. The unidentified error, that hindered progression of the task, was an obstacle that could be solved through the restart after a long period of tries to work around it.

6 Part 4: CAN-Ethernet-Gateway on Infineon AURIX TC297

This chapter focuses on the implementation of a CAN-Ethernet gateway.

6.1 Aim of the Practical Work

The objective of this work is to design, implement, and validate a bidirectional gateway that transparently links a classical CAN bus to a standard Ethernet network on an Infineon AURIX TriCore evaluation board. Requirements for the gateway are:

- {CAN → Ethernet}: listen on the CAN bus, encapsulate every accepted CAN frame in an IPv4/UDP (or alternatively RAW) payload, and forward it so that any IP-based device can interpret native CAN traffic
- {Ethernet → CAN}: accept UDP or RAW Ethernet frames that carry a properly formatted CAN payload, reconstruct standard CAN data frames and re-inject them onto the bus with correct arbitration timing

This compact overview frames the subsequent sections, which will give details about the hardware, concept and system architecture.

6.2 Hardware Overview: Infineon AURIX TC297

The Infineon TriCore microcontroller combines the capabilities of a microprocessor, microcontroller, and digital signal processor into one integrated platform, making it highly suited for automotive applications demanding real-time responsiveness and robust processing power.

Built around three 32-bit TriCore TC1.6P CPUs that run at up to 300 MHz, it couples up to 8 MB of ECC-protected flash with 728 kB on-chip RAM and a 128-channel DMA engine on a low-latency SRI cross-bar, so firmware can move payloads without stalling the cores. Gateway-relevant peripherals are integrated on-chip: a MultiCAN+ module with six independent CAN-FD-capable nodes and 10/100 Mbit/s Ethernet MAC that connects to an external PHY through MII or RMII. Because both engines share the same system bus, frames can be moved between CAN and Ethernet within deterministic microsecond budgets. A hardware security module, lock-step safety core, and redundant timer blocks complete the feature set required for ISO 26262 ASIL-D designs, making the TC297 a compact single-chip platform for the real-time CAN–Ethernet gateway realised in this project[2].

6.3 Theoretical Concept

A bidirectional gateway has to move information between two networks that sit at different layers of the ISO/OSI model: CAN is a controller-centric field bus defined entirely at layer 2, whereas Ethernet is normally used as the physical bearer for layer-3 protocols such as IPv4. The design therefore needs an encapsulation scheme that (i) preserves every CAN frame bit-exactly, (ii) introduces only constant and minimal latency, and (iii) passes unhindered through standard switching, routing and diagnostic equipment.

A classical CAN data frame carries an 11- or 29-bit identifier, a data length code (0-8 bytes in CAN 2.0, up to 64 bytes in CAN FD), the data field itself, plus bus-internal CRC and ACK bits. When the frame is moved to Ethernet only the semantic content must be preserved; the physical-layer bits are regenerated by the MULTICAN+ controller on retransmission. The project therefore serialises every CAN frame as a fixed-length record inside the UDP payload: four bytes little-endian identifier, one byte DLC, eight data bytes zero-padded, followed by one control byte whose individual bits flag extended identifiers and RTR frames. A short ASCII “magic” header (“ISO11898”) and a version byte at the start of the datagram allow a receiver to detect mis-alignment. Up to sixteen such records share one datagram, which keeps Ethernet utilisation high.

6.4 Software Implementation

The implementation starts with configuring the necessary hardware peripherals on the TriCore board, such as CAN and Ethernet modules. This is accomplished by using an initialization routine (Listing 1).

```
void initModules(void)
{
    /* Disable watchdog timers for stable operation */
    IfxScuWdt_disableCpuWatchdog(IfxScuWdt_getCpuWatchdogPassword());
    IfxScuWdt_disableSafetyWatchdog(IfxScuWdt_getSafetyWatchdogPassword
    ());

    /* Initialize status LEDs for debugging */
    initAliveLed();
    initBlinkyLed();
    initTxSuccessLed();
    initBufferFailLed();
    initRXETHLed();

    /* Initialize communication interfaces */
    initMultican();
    initEthernet();

    /* Initialize timing functions */
    initBlinkyTimer();

    /* Initialize the gateway module */
    initGateway();
}
```

Listing 1: Initialization of modules (Cpu0_Main.c)

This initialization function disables watchdog timers to ensure uninterrupted operation and sets up peripherals necessary for the gateway’s functionality, such as LEDs for visual debugging and status indication. It initializes the Multican and Ethernet modules, sets up timing mechanisms for periodic tasks, and prepares the gateway functionality.

The gateway logic is executed periodically within the main execution loop through the function `runGateway()`. This function orchestrates the two principal operations: the first function `forwardCanToEthernet()` handles translating CAN messages to Ethernet packets. The converting of CAN messages into Ethernet frames is shown in Listing 2.

```c
void forwardCanToEthernet(void)
{
    IfxMultican_Message rxMsg;
    /* Get a handle to the high-level CAN message object driver */
    IfxMultican_Can_MsgObj* rxMsgObjHandle = &g_multican.canRxMsgObj;

    /* Check if the dedicated RX message object has a new message */
    if (IfxMultican_Can_MsgObj_isRxPending(rxMsgObjHandle))
    {
        Ifx_CAN_MO* rxMo = IfxMultican_MsgObj_getPointer(g_multican.can.mcan, rxMsgObjHandle->msgObjId);

        IfxMultican_Can_MsgObj_readMessage(rxMsgObjHandle, &rxMsg);

        boolean isExtended = IfxMultican_MsgObj_isExtendedFrame(rxMo);
        boolean isRtr = FALSE;

        uint8 dlc = rxMsg.lengthCode;
        uint8 udpPayload[17];

        udpPayload[0] = 0x01;
        udpPayload[1] = 1;

        udpPayload[2] = (uint8)(rxMsg.id >> 0);
        udpPayload[3] = (uint8)(rxMsg.id >> 8);
        udpPayload[4] = (uint8)(rxMsg.id >> 16);
        udpPayload[5] = (uint8)(rxMsg.id >> 24);

        udpPayload[6] = isExtended;
        udpPayload[7] = isRtr;
        udpPayload[8] = dlc;

        uint8 can_data_bytes[8];
        can_data_bytes[0] = (uint8)(rxMsg.data[0] >> 0);
        can_data_bytes[1] = (uint8)(rxMsg.data[0] >> 8);
        can_data_bytes[2] = (uint8)(rxMsg.data[0] >> 16);
        can_data_bytes[3] = (uint8)(rxMsg.data[0] >> 24);
        can_data_bytes[4] = (uint8)(rxMsg.data[1] >> 0);
        can_data_bytes[5] = (uint8)(rxMsg.data[1] >> 8);
        can_data_bytes[6] = (uint8)(rxMsg.data[1] >> 16);
        can_data_bytes[7] = (uint8)(rxMsg.data[1] >> 24);

        for (int i = 0; i < dlc; i++)
        {
            udpPayload[9 + i] = can_data_bytes[i];
        }

        sendEthernetPacket(udpPayload, 9 + dlc);

        /*
         * FIX: Manually clear the Receive Pending flag.
         * This is the crucial step to prevent the same message from being read again.
         */
        IfxMultican_Can_MsgObj_clearRxPending(rxMsgObjHandle);
    }
```

```
}
```

Listing 2: CAN to Ethernet message translation (Gateway.c)

When a CAN message is received, this function checks for pending messages, reads the data into a structured format, and encapsulates it into a UDP packet. Metadata including the CAN identifier, message length, and payload are meticulously formatted. This encapsulation is critical for ensuring accurate data representation over Ethernet and allows diagnostic and monitoring tools to interpret the data correctly.

The implementation of the second operation `forwardEthernetToCan()`, that handles translating Ethernet packets to CAN messages, is implemented as shown in Listing 3.

```
void forwardEthernetToCan(void)
{
    uint8 ethPayload[17];
    uint16 payloadLen = receiveEthernetPacket(ethPayload, sizeof(ethPayload));

    if (payloadLen > 0)
    {
        // Packet validation
        if (payloadLen < 9 || ethPayload[0] != 0x01) { return; }
        uint8 dlc = ethPayload[8];
        if (dlc > 8 || payloadLen < (9 + dlc)) { return; }

        // Packet parsing
        uint32 can_id = ((uint32)ethPayload[2]) | ((uint32)ethPayload[3] << 8) | ((uint32)ethPayload[4] << 16) | ((uint32)ethPayload[5] << 24);
        boolean id_type = ethPayload[6];
        boolean frame_type = ethPayload[7];

        // Use the proven high-level transmit function.
        // This will only forward standard data frames as per the TX message object's initial configuration.
        if (frame_type == 0 && id_type == FALSE)
        {
            uint32 dataLow = 0;
            uint32 dataHigh = 0;
            for (int i = 0; i < dlc; i++)
            {
                if (i < 4) { dataLow |= ((uint32)ethPayload[9 + i] << (i * 8)); }
                else { dataHigh |= ((uint32)ethPayload[9 + i] << ((i - 4) * 8)); }
            }

            // Create the CAN message with the ID and payload from the UDP packet
            IfxMultican_Message canMsg;
            IfxMultican_Message_init(&canMsg, can_id, dataLow, dataHigh, (IfxMultican_DataLengthCode)dlc);

            // Send the message using the function we know works reliably
            transmitCanMessage(&canMsg);
        }
```

```
    }
}
```

Listing 3: Ethernet to CAN message translation (Gateway.c)

Upon receiving an Ethernet packet, this function validates the packet structure and contents, ensuring it meets specific criteria such as minimum length and identifier correctness. It then extracts the payload, reconstructs the CAN message identifier and data fields, and transmits this data over the CAN network. This detailed parsing and validation ensure that only valid data is transmitted, preventing erroneous or corrupted data from entering the CAN system.

In the implemented firmware every CAN frame is carried inside a single UDP datagram whose payload is a compact, variable-length record of 9–17 bytes. The preceding Ethernet, IPv4 and UDP headers are generated by `sendEthernetPacket()` and stripped again by `receiveEthernetPacket()`. On transmit `forwardCanToEthernet()` fills these fields and hands the buffer to the Ethernet driver, thereby incurring a constant per-frame overhead of 9 bytes and avoiding dynamic memory allocations. On reception the reverse routine validates the synchronisation byte and DLC before reconstructing the CAN message.

In the opposite direction, CAN frames are captured, structured, and transmitted via Ethernet as UDP packets using the `sendEthernetPacket` function, which handles packet construction and transmission management while ensuring data integrity and frame correctness.

The Construction and Sending of the packets were realized with a Python-based UDP test script to validate the Ethernet-to-CAN transmission functionality (Listing 4). This script enables controlled injection of CAN message frames into the system from a host PC over a UDP network and provided a flexible and repeatable way to validate standard data frame transmissions, confirming the correct end-to-end behavior from the host PC to the CAN bus via UDP encapsulation.

```
# --- Build the CAN message payload ---
# Version: 1, Channel: 1
payload = bytearray([0x01, 0x01])

# CAN ID: 0x7DF (as a 32-bit little-endian integer)
can_id = 0x7DF
payload.extend(can_id.to_bytes(4, 'little'))

# ID Type: 0 (Standard), Frame Type: 0 (Data), DLC: 8
payload.extend([0x00, 0x00, 0x08])

# Data: [0x02, 0x10, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00]
can_data = bytearray([0x55, 0x44, 0x50, 0x54, 0x4F, 0x43, 0x41, 0x4E])
payload.extend(can_data)

# --- Send the UDP packet ---
print(f"Sending {len(payload)} bytes to {DEVICE_IP}:{UDP_PORT}")
print(f"Payload (hex): {payload.hex().upper()}")

try:
    # Create a UDP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
23
24      # If a specific source IP is set, bind the socket to it
25      if SOURCE_IP != "0.0.0.0":
26          print(f"Using network adapter with IP: {SOURCE_IP}")
27          # Binding to port 0 tells the OS to pick any available ephemeral
        port
28          sock.bind((SOURCE_IP, 0))
29      else:
30          print("Using default network adapter chosen by OS.")
31
32      # Send the data
33      sock.sendto(payload, (DEVICE_IP, UDP_PORT))
34      print("Packet sent successfully!")
35  except Exception as e:
36      print(f"Error sending packet: {e}")
37  finally:
38      sock.close()
```

Listing 4: UDP Packet Sending

The script constructs a synthetic CAN frame embedded within a UDP payload conforming to the format expected by the gateway firmware. The structure of the payload follows the implemented protocol in `Gateway.c`, where the CAN message is encoded in the following order: a version byte, a channel identifier, a 32-bit CAN ID in little-endian byte order, frame metadata (ID type, frame type, data length code), and finally the actual CAN data payload. In this specific example, the CAN ID is set to `0x7DF`, which corresponds to a diagnostic request in the automotive OBD-II standard. The data segment contains eight bytes with the ASCII representation of "UDPTOCAN," which assists in tracing and distinguishing injected messages during runtime analysis and debugging.

The socket communication is implemented using Python's `socket` module. A UDP socket is created and optionally bound to a specified source IP address, which allows manual selection of the desired network interface. This feature is particularly relevant in multi-adapter systems where deterministic routing of test packets is required. The use of port `60000` aligns with the configuration defined in the embedded gateway firmware (`ETH_SRC_PORT` in `Ethernet_Example.h`) ensuring compatibility.

6.5 Problem Report

During the implementation using the Infineon AURIX TriCore microcontroller, several technical challenges were encountered that significantly influenced the final software structure and functionality.

One of the primary issues arose from the Ethernet communication subsystem. Specifically, system instability and repeated microcontroller crashes were observed when attempting to handle incoming Ethernet frames using interrupt-driven mechanisms. Although interrupt-based handling is typically preferred for its efficiency and responsiveness, its application in this context led to non-recoverable system states. The cause of these failures could not be conclusively identified from the driver-level code provided in the development libraries. Furthermore, the extensive official documentation for the Ethernet module—spanning over 600 pages—did not offer a clearly accessible explanation

or guidance regarding this failure mode. Consequently, Ethernet packet handling was implemented using polling techniques instead. While less efficient, polling ensures system stability and predictable behavior in this application.

A second limitation was encountered in the handling of CAN messages, specifically in the attempt to manually construct a universal CAN message capable of transmitting remote transmission requests (RTR frames) through a low-level access approach. Although the implementation followed the structure of the CAN message object initialization sequence and replicated known patterns from standard data frame handling, the constructed messages failed to execute as intended. The relevant section of code in `Gateway.c`, which attempted this advanced message creation logic, had to be commented out and remains non-functional. As a result, the current system is restricted to processing and forwarding only standard data frames from Ethernet to CAN, without support for RTR frames.

6.6 Result

The result could be confirmed upon testing and verifying with Wireshark. In Figure 5,

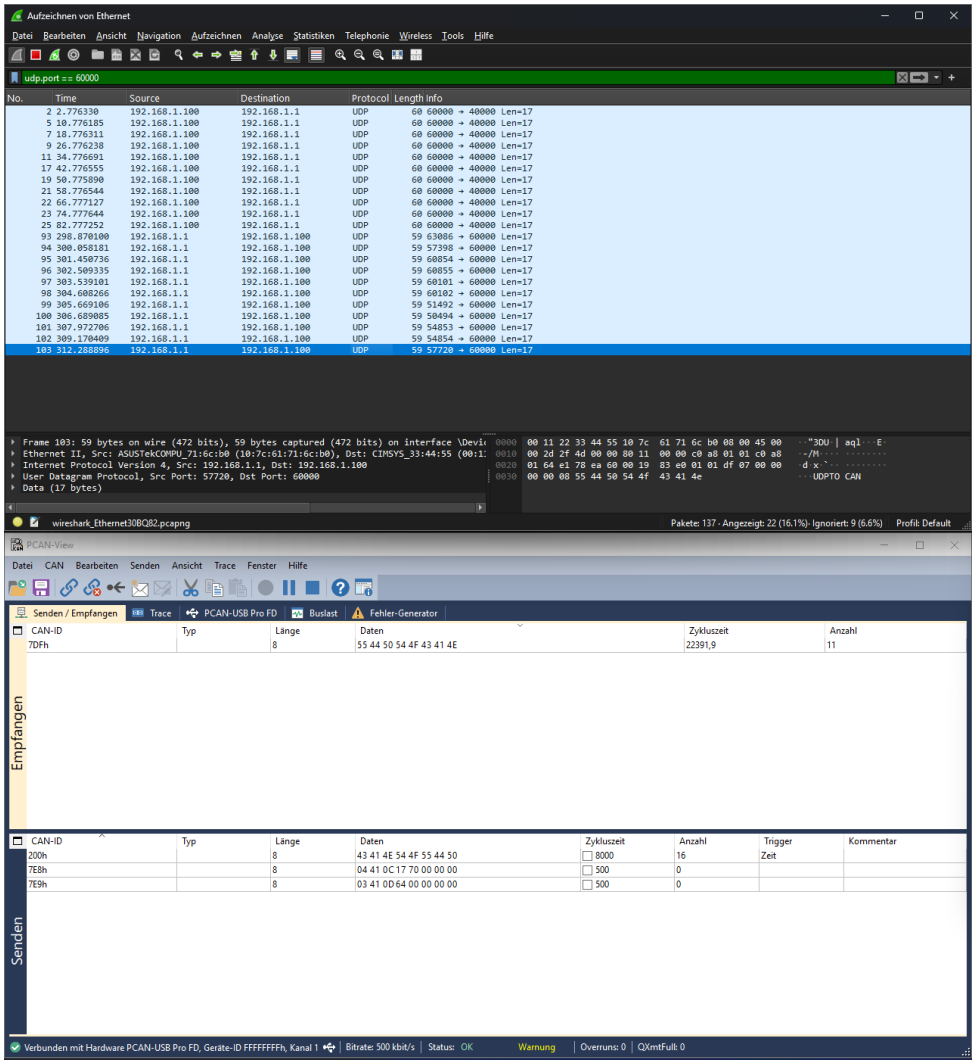

Figure 5: Ethernet to Can

Wireshark records multiple UDP packets originating from a PC (192.168.1.1) and addressed to the microcontroller at 192.168.1.100 on port 60000. Each UDP payload is

exactly 17 bytes long, conforming to the defined CAN message encapsulation format. The payload contains a recognizable ASCII sequence "UDPTOCAN," which serves as a test marker. Simultaneously, the lower window (PCAN-View) shows that the CAN message with ID `0x7DF` and the correct payload bytes has been successfully received and decoded by a connected CAN interface. This demonstrates that the embedded function `forwardEthernetToCan()` correctly extracted and transmitted the CAN message based on the incoming UDP frame.

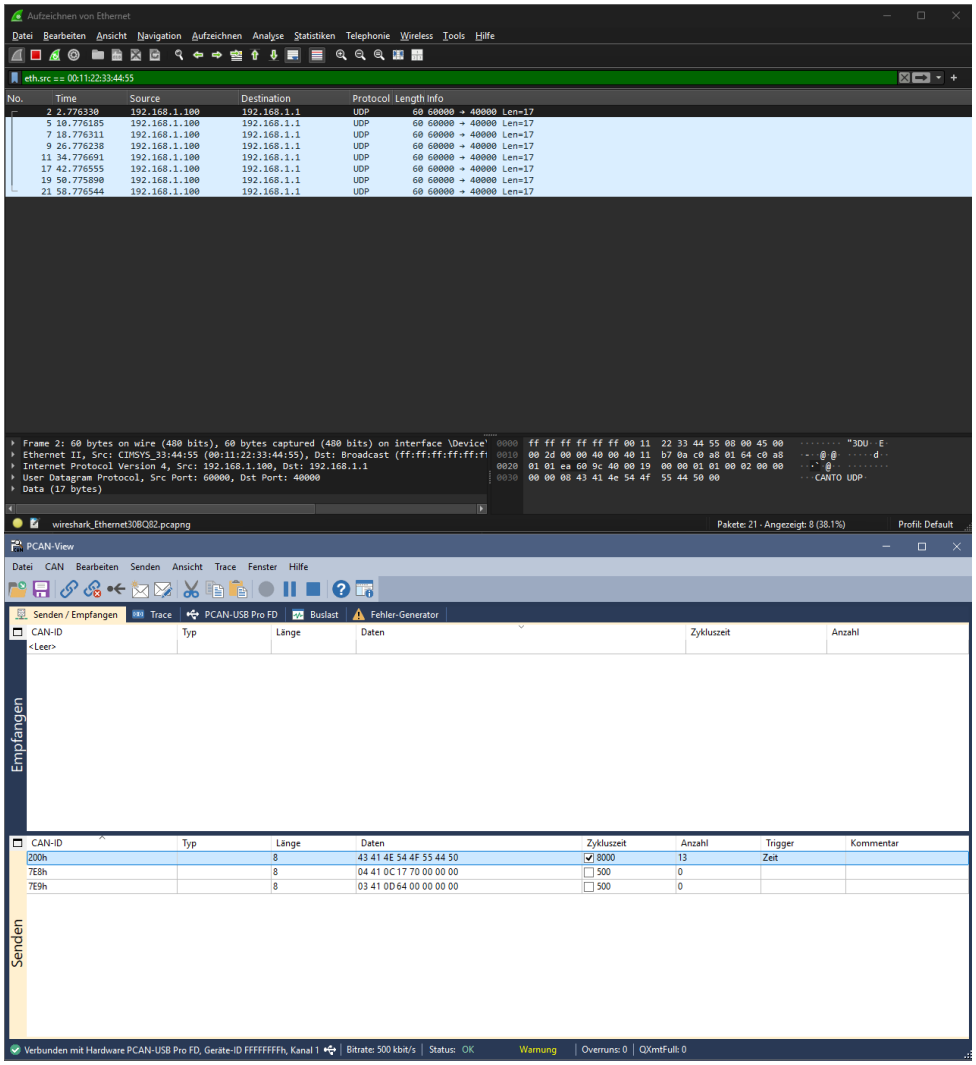

Figure 6: Can to Ethernet

Figure 6 shows the reverse communication path. Here, the CAN message with ID `0x200` and data payload corresponding to the ASCII string "CANTOUDP" is transmitted from the CAN bus. Wireshark captures its Ethernet encapsulation, showing correct transmission from the microcontroller to a broadcast destination (MAC: FF:FF:FF:FF:FF:FF) on UDP port 40000. The embedded function `forwardCanToEthernet()` has successfully serialized the CAN frame into a valid UDP packet and injected it into the Ethernet stack.

Together, these bidirectional traces confirm that both gateway directions—Ethernet to CAN and CAN to Ethernet—function as intended. The protocol is correctly formed, hardware peripherals are synchronized, and application-layer behavior is predictable and

reliable. This validation confirms that the goals of the project were achieved and a fully operational CAN-Ethernet gateway with UDP support was realized on the Infineon TriCore platform.

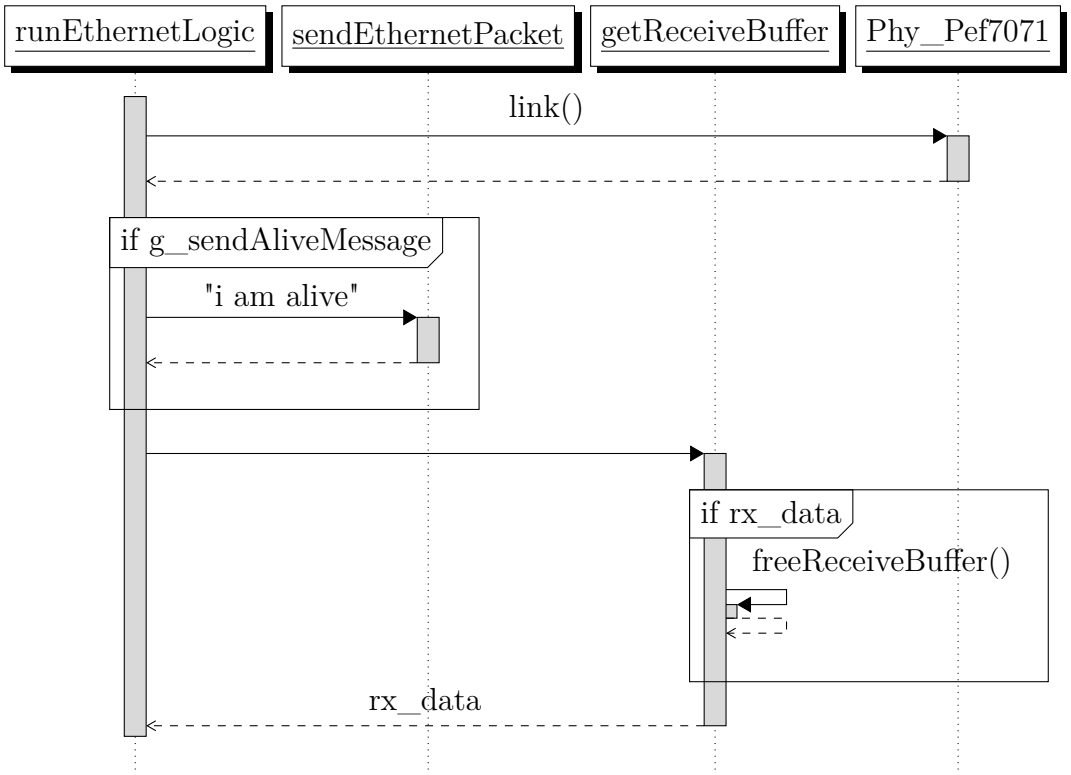

Figure 7: UML Diagramm 1: Ethernet Rx/Tx Logic

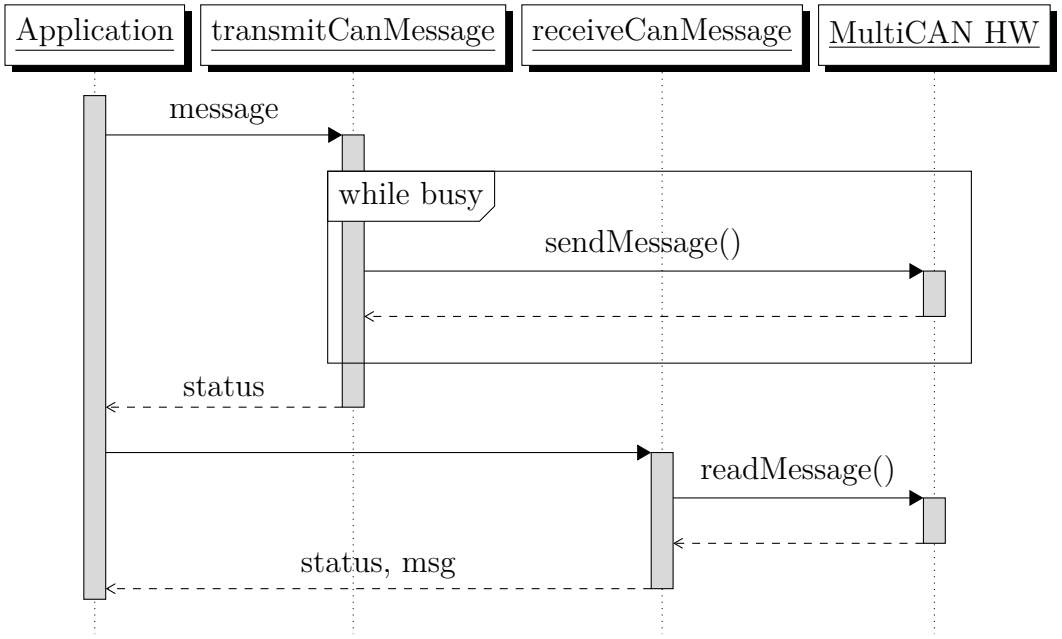

Figure 8: UML Diagramm 2: CAN Rx/Tx Logic

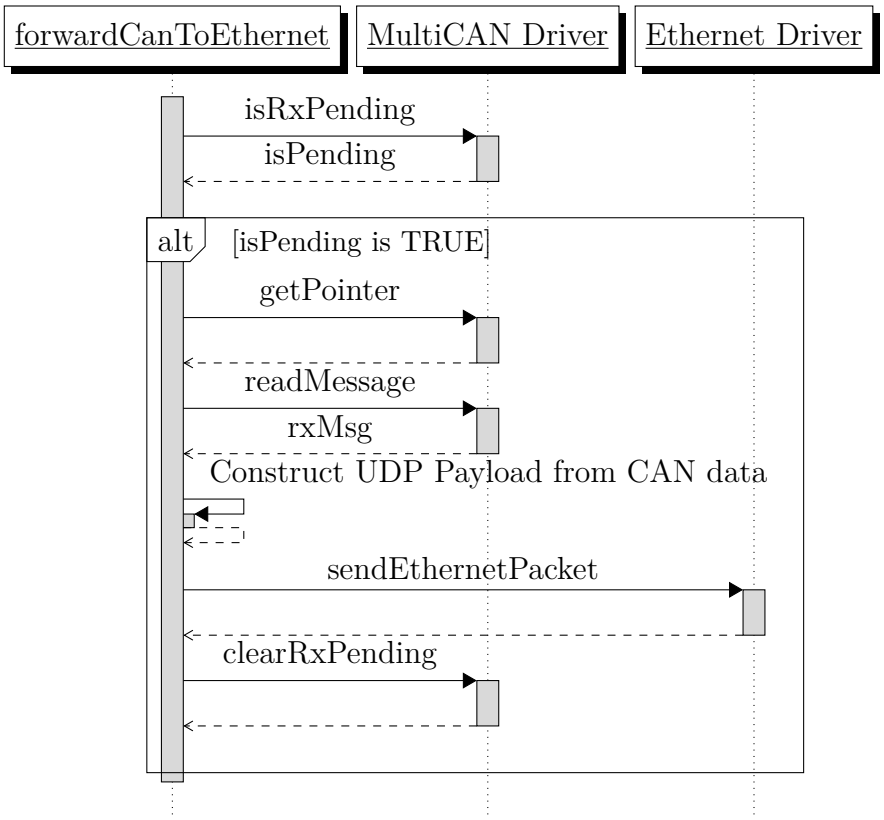

Figure 9: UML Diagramm 3: CAN to Ethernet Message Forwarding

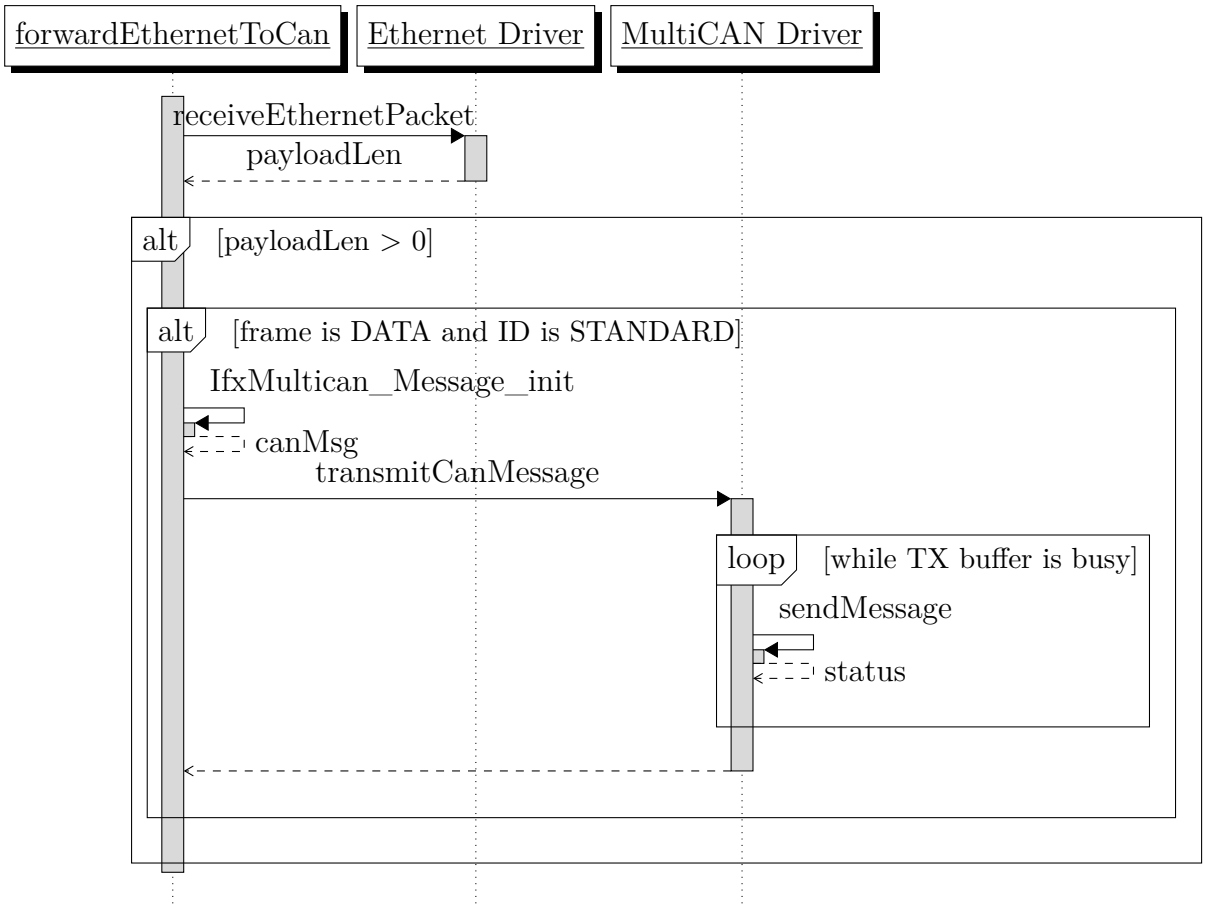

Figure 10: UML Diagramm 4: Ethernet to CAN Message Forwarding

References

[1] T. Engineering.

[2] Infineon.