POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Design and Implementation of Ethernet-Based Real-Time SPI Protocol Management using dSPACE HIL Simulator and STMicroelectronics Boards

Supervisors:

Prof. Massimo Violante

Giuseppe Selva

Candidate

Usama Islam

July 2025

Acknowledgements

I want to thank **Kineton s.r.l.** for giving me the opportunity to carry out my thesis in their esteemed organization. Their furnishing with minimum resources and infrastructure played a vital role in the successful completion of my project. Moreover, I would like to thank my manager, **Giuseppe Selva**, for his continuous support, guidance, and patience. His insightful advice and enthusiasm helped a lot towards my success and learning in the project.

Furthermore, I would like to express my gratitude to my academic supervisor, Professor **Massimo Violante**, for his most valuable guidance and advice, which helped shape and refine this thesis.

Abstract

Real-time Hardware-in-the-Loop (HIL) simulation is crucial for the comprehensive testing and validation of automotive Electronic Control Units (ECUs) before they are integrated into a vehicle. Robust HIL environments can easily interface with ECUs over Ethernet; however, critical components like Battery Management Systems (BMS) and sensors/actuators often rely on short-distance, low-latency protocols such as SPI. This thesis presents the design, implementation, and verification of a real-time communication bridge between Ethernet-based UDP protocols and full-duplex SPI interfaces, addressing the challenge of integrating such SPI-based devices into Ethernet-based HIL simulators. Implemented on STM32F207ZG Nucleo boards configured in a Master-Slave topology, the Master board interfaces with an external HIL environment (dSPACE or a Python-based prototype) via Ethernet (UDP protocol). The Master board translates received UDP payloads into SPI transactions for the Slave board. The Slave board processes these SPI commands/data from the Master and returns the corresponding response.

The development started with a Python-driven UDP-SPI echo prototype and upgraded into a reliable communication system, along with the key features being a custom SPI protocol with robust framing, software-based CRC-8 for data validity, and support for variable-length/type data transactions. The system utilizes DMA-based SPI transfers to achieve a deterministic 1-ms cycle time for SPI transactions. An important enhancement was the implementation of circular DMA on the Slave board for continuous data handling and response generation. On the Master, SPI transactions were triggered by incoming UDP packets and are supported by a polling mechanism for continuous data streams from the Slave.

The implemented Ethernet-SPI Bridge was successfully integrated with the dSPACE SCALEXIO platform. Distinct Simulink models for echo verification and command-based sensor streaming were configured via dSPACE ConfigurationDesk. These models were deployed as real-time applications and controlled using dSPACE ControlDesk to generate UDP dynamic values or specific commands and subsequently receive/display the corresponding echo data or emulated sensor responses from the STM32 system. This hardware-software co-simulation showed effective command handling for various sensor modes (Idle, Speed, Gyro, Accel) and accurate real-time data feedback to the HIL environment.

The resulting solution demonstrates a stable and responsive Ethernet-SPI bridge capable of sustained 1-ms data exchange in the context of HIL simulation, with error detection and recovery mechanisms. This research provides a practical and flexible framework for real-time sensor emulation that can be directly applicable to industrial automotive HIL testing and lays a strong foundation for future integration with automotive ECUs and the development of ISO-compliant communication modules.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context and Motivation

Over the past few decades, the automotive industry has made vast steps in the improvement of communication between Electronic Control Units (ECUs), sensors, and actuators. In modern systems, all these must operate in tightly coordinated timing frameworks, as many of them are communicating across both physical and logical layers. Serial Peripheral Interface (SPI) is widely used for the short-range communication between microcontrollers and sensors/actuators because of its simplicity and low-latency characteristics. However, in distributed automotive systems and especially in autonomous electric vehicles, the communication generally begins in Ethernet or User Datagram Protocol (UDP)-based protocols at higher system levels and must be translated into lower-level bus protocols such as SPI or Controller Area Network (CAN) at the embedded layer.

Hardware-in-Loop (HIL) platforms such as dSPACE SCALEXIO play a vital role in the development and testing of such systems. At Kineton s.r.l., for instance, there is extensive utilization of the dSPACE HIL simulator, where configuring Ethernet-based test scenarios is a well-established practice. However, a growing future requirement involves the comprehensive testing of various automotive Electronic Control Units (ECUs), such as Battery Management Systems (BMS), which predominantly communicate via the SPI protocol. Currently, directly interfacing numerous SPI-based ECUs with the existing Ethernet-centric HIL infrastructure presents a challenge. So, creating a reliable and flexible way to connect HIL-generated Ethernet (UDP) data with hardware that uses SPI is not just a technical challenge but also a key step for improving future testing, especially when aiming for real-time cycles of 1-ms. The work done in this thesis directly addresses this need.

1.2 Project Objective

The primary and foremost goal of this thesis is to design and implement a reliable full-duplex communication system between an SPI-based embedded system using STM32 microcontroller boards and a dSPACE SCALEXIO HIL simulator. Specifically, the aim of this thesis is:

- To develop and validate a foundational real-time Ethernet-to-SPI communication gateway, demonstrating its feasibility and performance for HIL testing, thereby providing a scalable framework to facilitate future integration and testing of SPI-interfaced automotive ECUs, such as BMS, within existing Ethernet-based dSPACE HIL environments.
- Establish a real-time UDP-based communication between the STM32 microcontroller board with both the Python script, which is a prototype system, and packets generated by HIL-simulator.
- Implement a full-duplex SPI protocol between two STM32 Microcontroller boards (Slave and Master) with robust framing, variable-length and variable-type data support, and software-based CRC-8 error detection.
- Create a closed-loop where commands (e.g., speed values) transmitted via UDP are passed on to the slave as SPI and matching feedback is sent back via UDP, creating a round-trip communications path.
- Offer scalability in the protocol through provision for differing data types and commands, for instance, float and integer types of value, and replicating responses of diverse sensor or actuator responses.
- Everything must be brought together using Simulink and dSPACE ControlDesk to simulate sensor inputs, generate real-time UDP packets, and present responses, providing a platform for subsequent integration with real ECUs or automotive sensors.

The prototype is a modular and scalable communication stack that can be used for SPI or other such protocols, enabling robust pre-validation in automotive software-in-the-loop and hardware-in-the-loop development cycles.

1.3 Structure of the work

The thesis is organized as follows:

- Background and related work that addresses the theoretical background and past work done in the domain relevant to the thesis. It includes the overview of the protocols that are used in the automotive systems, the behavior of real-time systems, SPI technology, and the existing work done involving STM32 microcontrollers in HIL environments.
- System Architecture presents a broad hardware and software outline of the proposed system. System Architecture covers STM32-based SPI communication setup, Ethernet

integration through LWIP middleware, and use of dSPACE SCALEXIO with Simulink and dSPACE ControlDesk for simulation and control.

- Implementation and methodology adopted, which explain the development phase, starting from initial UDP communication using Python script till the full-scale development involving a full-duplex SPI interface, which involved CRC-based framing. It also includes how dSPACE ControlDesk and Simulink were integrated into the embedded systems.
- Testing approaches adopted and the results obtained in order to do the system analysis are also covered. It includes results of Python simulations, Wireshark packet capture, and performance testing done on a real-time simulator using dSPACE ControlDesk.
- Conclusion and Future Work summarizes the key results obtained in the project, evaluation of the system's abilities, and proposes potential improvements, such as additional command types, timing constraints, and integration with real automotive sensors/ECUs and SPI hardware transceivers.

Chapter 2

Background and Related Work

2.1 Automotive Communication Protocols

Over the years, modern automotive systems have become heavily reliant on robust, low-latency communication protocols to ensure that the coordination between ECUs, sensors, and actuators is smooth. In this section, the primary focus is to cover the communication protocols that are relevant to the work done in the thesis.

2.1.1 Ethernet Protocol

Ethernet was originally developed in 1973 at Xerox PARC by Robert Metcalfe and his team to provide high-speed communication between computers over a shared medium [1]. Leveraging the ALOHAnet system, Ethernet evolved the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol to manage the transmission of data and avoid collisions. The initial Ethernet operated at 2.94 Mbps over thick coaxial cable. Digital Equipment Corporation, Intel, and Xerox (DIX) in 1980 released the initial Ethernet specification that was later standardized by the IEEE in 1983 as IEEE 802.3.

Over the decades, Ethernet has been thoroughly enhanced to meet growing demands for higher bandwidth and more reliable communications [2].

- 10BASE-T (1990): Took up twisted-pair cabling, facilitating easier installations.
- 100BASE-TX (1995): Known as Fast Ethernet, increasing speeds to 100 Mbps.
- 1000BASE-T (1999): Gigabit Ethernet over twisted-pair cables.
- 10GBASE-T (2006): It ensured 10 Gbps speeds running over copper cable.

- 100GBASE-R (2010s): It increased the speeds up to 100 Gbps and above running over fiber optics cable.

These innovations have played a key role in supporting intensive data applications and have ensured that Ethernet is one of the most widely used mediums in networking today.
Ethernet generally operated at the lower layers of Open Systems Interconnection (OSI) model i.e. Layer - 1, which is the physical layer, and Layer – 2, which is the data link layer.

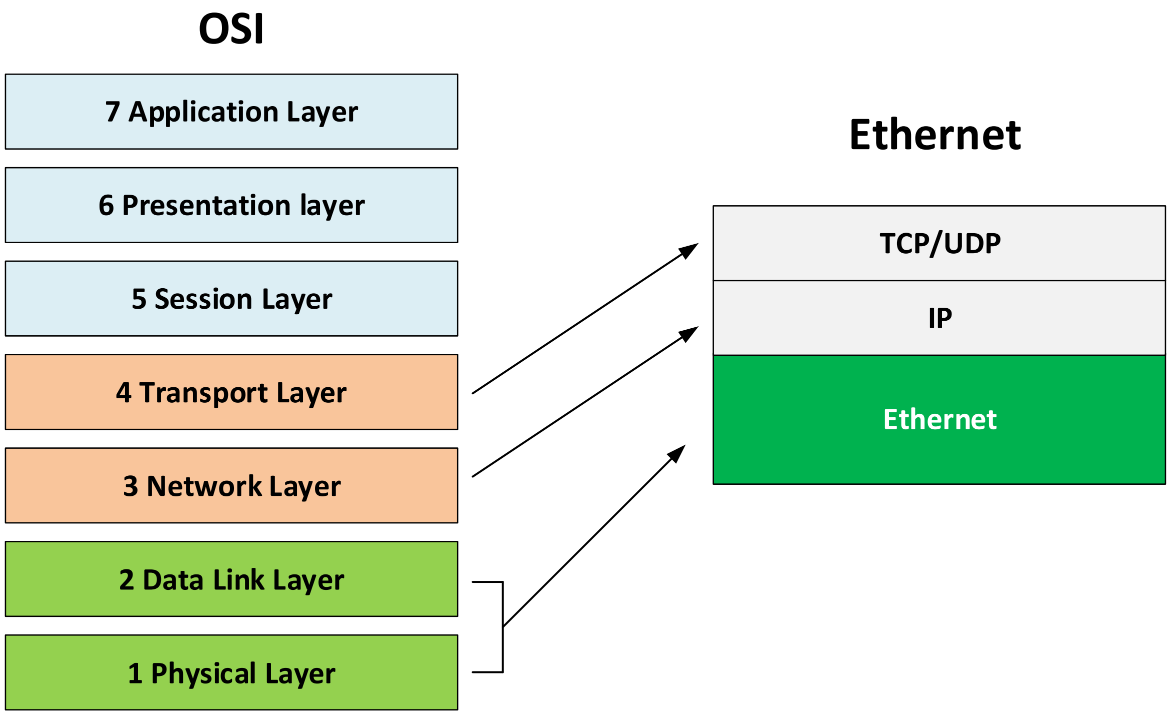


Figure 2-1: OSI Layer and Ethernet [3]

a) <u>Physical Layer:</u>

The main concern of this layer is transmission of raw bits over the physical medium. This layer defines the electrical characteristics, transmission medium, bit encoding/decoding methods, cables, and connector types.
The most common cables used these days are twisted pair cables (CAT5, CAT6 and CAT7), and the latest one is CAT8 cable, which can operate up to the speed of 40 Gbps. These Ethernet twisted pairs use RJ-45 connectors to establish the link. Fiber optic cables are also used, which are capable of providing high-performance long-distance communication. These are widely used by data centers and for Wide Area Network (WAN).
Ethernet PHY (Physical Layer Transceiver) is a dedicated chip that is used to handle functions required for the functioning of this physical layer. The functions carried out by PHY includes modulating/demodulating the electrical signals, encoding/decoding the bits, and auto-negotiation, which includes selection of the speed and duplex, i.e. Half Duplex, Full Duplex. The PHY

communicates with Medium Access Controller (MAC) via a digital interface, which generally comprises [2]:

- Media Independent Interface (MII), which is typically used for data transfer rates of 10 Mbps and 100 Mbps.
- Reduced Media Independent Interface (RMII), which is used for data transfer rates of 10 Mbps and 100 Mbps, but they have a reduced pin count compared to MII.
- Gigabit Media Independent Interface (GMII), which is typically used for data transfer rates of 1000 Mbps.
- Reduced Gigabit Media Independent Interface (RGMII), which is typically used for data transfer rates of 1000 Mbps.

b) <u>Data Link Layer:</u>

The OSI model's Layer 2, the Data Link Layer, is tasked with the task of secure node-to-node data transfer over a physical network. In Ethernet, that means the delivery of packets between two devices using MAC addressing and framing operations. Key data link layer functions are:

- Framing: Wrap the payload (e.g., UDP, TCP data) with MAC addresses, length/type, and Frame Check Sequence (FCS) for error detection.
- Addressing: Uses MAC addresses (48-bit unique hardware addresses) to identify the source and destination on a LAN.
- Error Detection: Uses a Frame Check Sequence (usually CRC-32) to detect any transmission errors.
- Media Access Control: It employed CSMA/CD in the classic Ethernet (shared media). In modern switched full-duplex Ethernet, contention is avoided and frames are transmitted forward directly.

According to the IEEE 802 standard, the data link layer is divided into two sublayers:

i. Logical Link Layer (LLC):

Logical Link Control (LLC) Sublayer (IEEE 802.2): Originally designed to provide a common interface to higher-layer protocols and services like connectionless or connection-oriented communication. In IP-based networks, a minimal LLC is used, or it's avoided by the EtherType field.

ii. Media Access Control (MAC):

Media Access Control (MAC) Sublayer (IEEE 802.3): Framing (encapsulating data into Ethernet frames with MAC addresses and FCS), MAC addressing, and access to the physical medium management (e.g., CSMA/CD in old Ethernet or simply passing frames to the PHY in switched full-duplex systems).

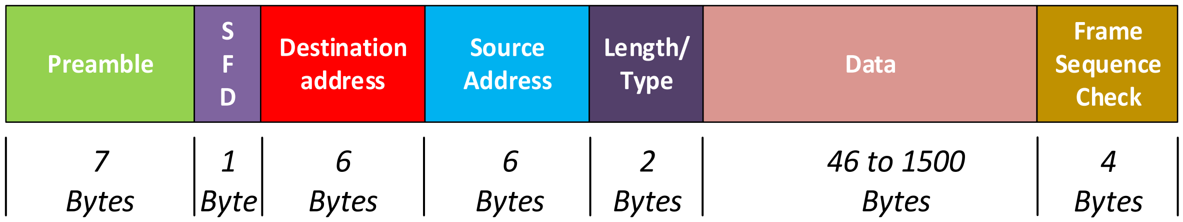


Figure 2-2: Ethernet Frame Format [4]

- Preamble: These are alternating 1s and 0s used for the clock synchronization.
- Start Frame Delimiter (SFD): This is marked at the end of the preamble and before the start of the frame content.
- Destination Address: It is the MAC address of the recipient device.
- Source Address: It is the MAC address of the sending device.
- Length/Type: It has a threshold for the values; if the value is $\geq$ 1536 (0X0600), it is an EtherType field, which indicates the protocol of the encapsulated payload. If the value is $\leq$ 1500, it is the length field, which indicates the length of the payload.
- Data: This is the size of the actual data being transmitted. If the data is less than 46 bytes, padding is done in order to meet the minimum size requirements. The maximum size is 1500 bytes.
- Frame Sequence Check: It is a 32-bit Cyclic Redundancy Check (CRC), which is calculated over the frame (starting from Destination MAC to Data) for error detection.

c) *Network/Internet Protocol layer*

The Internet Protocol (IP) is the fundamental communications protocol in the Internet protocol suite for delivering datagrams from one network to another. It has only one purpose, which is to deliver packets of data between a source host and a destination host based on their IP address. IP resides at the Network Layer (Layer 3) of the OSI model and the Internet Layer of the TCP/IP model. It is the backbone of the internet and most modern private networks, allowing inter-networking of diverse physical networks, like Ethernet. The key concepts of this layer are depicted in the table below:

Table 2-1: Network Layer Features

Feature	**Description**
IP Address	This is the 32-bit address (IPv4) identifying source/destination (e.g., 192.168.1.100)
Subnet Mask	It defines the network boundary (e.g., 255.255.255.0)
Gateway	It is the route to other networks or subnets
Time-to-Live (TTL)	This is essential as it prevents packets from looping forever

Protocol Field	It is used for the indication of next layer i.e. e.g., 0x06 for TCP, 0x11 for UDP
Checksum	Header level error detection

The Transport Layer (Layer 4) of the OSI model has the responsibility of providing end-to-end communication between applications on various networked systems. It provides mechanisms for:

- Segmentation of application data into packets
- Port addressing to differentiate services
- Voluntary flow control, reliability, and error management

Two major protocols used at this level are UDP (User Datagram Protocol) and TCP (Transmission Control Protocol)—both are run on top of the IP protocol (Layer 3) and below the application logic. These protocols are defined in the next subsections.

2.1.1.1 User Datagram protocol (UDP)

UDP is a stateless, connectionless, lightweight transport protocol defined in RFC 768. It is optimized for small overhead and low latency and hence suited for real-time and embedded systems applications.

a) Key Characteristics

Table 2-2: UDP Features

Feature	**Description**
Connectionless	There is no prior handshake before the data is sent. Each UDP datagram is an independent unit.
Unreliable	UDP does not guarantee that the datagram will arrive at the destination. It also doesn't guarantee that the data will arrive in order or will arrive with duplication. There are no acknowledgements or retransmissions in UDP protocol.
Datagram-Oriented	UDP preserves message boundaries. Application messages are sent as independent datagrams. When a datagram is received, it's the whole message that the application dispatched (except perhaps if IP-framed, but UDP itself deals with the whole messages).
No Flow Control	UDP never controls how fast data is sent from the sender to prevent overwhelming the receiver.
No Congestion Control	UDP never responds to network congestion by slowing down transmission. This can be an issue for overall network wellness if not controlled by the application.

Low Overhead	UDP adds a very small header to the data.

b) *UDP header*

The UDP header has a fixed size of 8-bytes, and it comprises the following segments shown in Figure 2-3: UDP Header below:

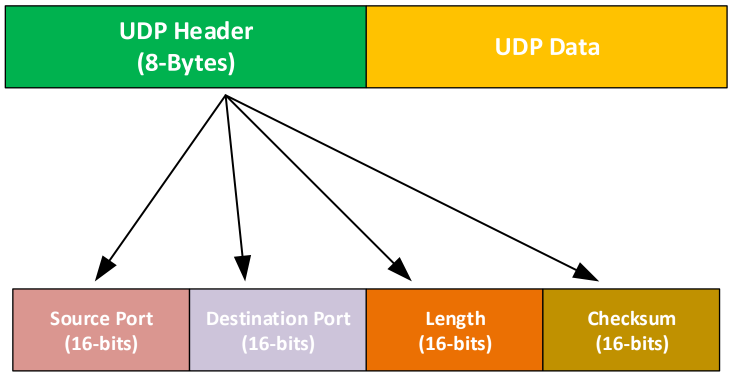


Figure 2-3: UDP Header [5]

- Source Port (16-bits): Identifies the sending process application within the source host. Optional; can be zero if responses are not received by the source. Port numbers range from 0 to 65535, and port number 0 is reserved.
- Destination Port (16-bits): It corresponds to the receiving application process on the target host. It is a required field. Its port range is 0 to 65535, and port number 0 is reserved.
- Length (16-bits): Length of the UDP data plus the UDP header in bytes. Minimum of 8 (for an empty UDP datagram).
- Checksum (16-bits): An optional error-detection facility (in IPv4; mandatory when IPv6 is used). It encompasses the UDP header, the UDP data, and an IP "pseudo-header" (destination/source IP addresses, protocol number, and UDP length). When the IPv4 facility is not utilized, it is set to zero. When the facility is calculated and is non-zero, zero is transmitted as all ones (0xFFFF).

c) *Working Principle*

When an application sends data through UDP, UDP simply takes the data and attaches its 8-byte header (with appropriate port numbers, length, and optional checksum) and then passes on the resulting datagram to the IP layer to be encapsulated and transmitted. The communication flow of UDP is shown in Figure 2-4: UDP Communication Flow below:

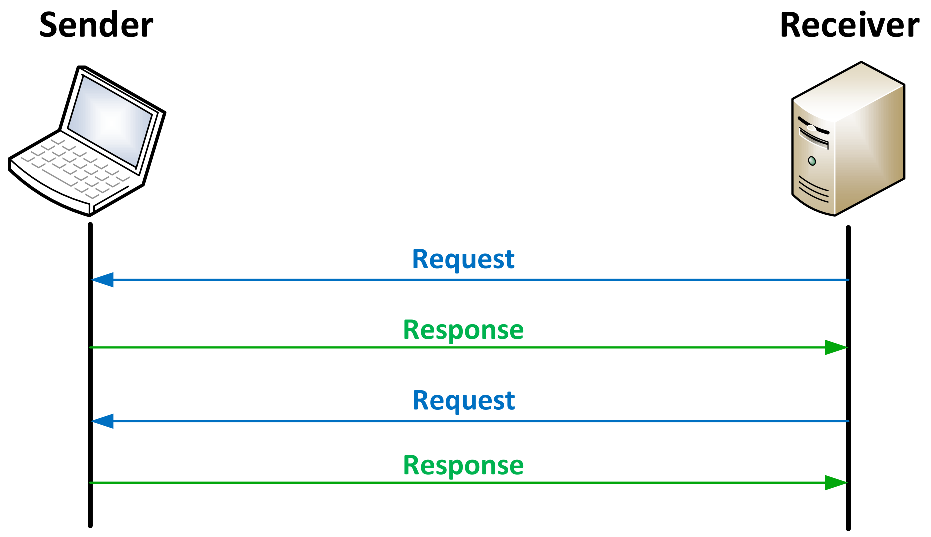


Figure 2-4: UDP Communication Flow [5]

d) *Advantages and Disadvantages*

Table 2-3: UDP Advantages vs. Disadvantages

Advantages	**Disadvantages**
High speed and low latency because of no handshake/connection setup	Unreliable as it does not have a mechanism for acknowledgment reception. Packet loss, duplication, and reordering must be handled by the application.
Low header size therefore requires less processing time	No Flow or Congestion control, which can lead to network problems if application sends the data too quickly without taking into account the capacity of the receiver or the network conditions
It has a simpler design, so it is easy to implement	
Suitable for Multicast and Broadcast transmission as it has a connectionless nature	
Gives the application full control over timing and reliability, which is beneficial for real-time systems	
Low Overhead	

2.1.1.2 Transmission Control Protocol (TCP)

TCP is a connection-oriented and reliable protocol specifically designed for guaranteed data transmission. Defined in RFC 793, it ensures that the packets are delivered in the correct order, retransmitted in case of loss/no acknowledgment and checked for integrity.

a) Key Characteristics

Table 2-4: TCP Features

Feature	**Description**
Connection-Oriented	There is a proper handshake between two TCP entities, using a “Three-way handshake” before one can send the data. The connection is terminated explicitly once the data has been transferred.
Reliable Delivery	TCP uses sequence numbers and acknowledgments (ACKs) to ensure that the data delivery is done properly from one end to another. The data that is lost or corrupted is retransmitted.
Ordered Delivery	TCP ensures that the data being delivered to the receiving application is in the same order as it was sent by the sending application. It uses sequence numbers, which reorder the segments that are out of sequence.
Flow Control	TCP uses a sliding window method, which prevents a fast sender from overwhelming a receiver that is slow in response. The receiver notifies with its available buffer space (receive window)
Congestion Control	TCP uses algorithms (e.g., slow start, congestion avoidance, fast retransmit, and fast recovery) to detect and respond to network congestion and reduce its sending rate to avoid overloading the network.
Byte-Stream Oriented	TCP treats data as a continuous stream of bytes rather than as an individual message or datagram. Message boundaries of an application are not necessarily preserved upon arrival unless the application layer employs framing of its own.

b) TCP Header

The TCP header has a minimum size of 20 bytes and a maximum size of up to 60 bytes. It comprises the following segments as shown in Figure 2-5: TCP Header below:

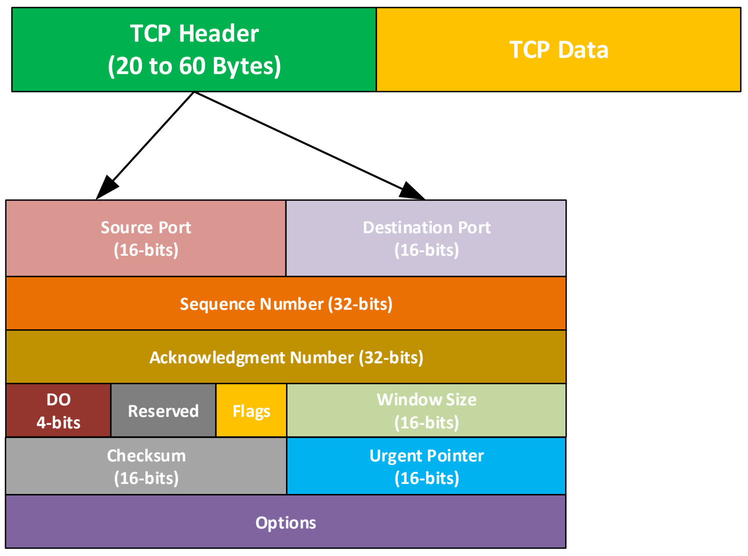


Figure 2-5: TCP Header [6]

- Source Port (16-bits): Identifies the sending application process.
- Destination Port (16-bits): Identifies the receiving application process
- Sequence Number (32-bits): It specifies the amount of data that is sent during the TCP session. The sequence number is used to identify each byte of the data and ensure that the data is delivered in order and there are no duplications as well.
- Acknowledgement Number (32-bits): When acknowledgement flag is set, this field holds the value of the next sequence number. This field is used by the receiver to acknowledge the data that has been received and subsequently request the next data.
- Data Offset (4-bits): This field is used to determine the size of the TCP header. It refers to where the data starts.
- Reserved (3-bits): it must be set to zero and is reserved for future uses.
- Flags (9-bits): it contains several controlling bits that are useful for indicating the states of a TCP connection. These flags are Urgent Pointer Field (URG) which indicates urgent data, Congestion Window Reduced (CWR) which is sender reduced rate, ECE (ECN-ECHO) which means the receiver contains a packet with ECN congestion experienced set bit, Acknowledgment field significant (ACK) which is that the segment carries an acknowledgment, Push Function (PSH) which is to command the receiver to deliver the data to application in a rush, Reset the Connection (RST) which is used to reset the connection, Synchronize Sequence Number (SYN) which is used to synchronize a connection and Finish (FIN) which is used to close the connection when there is no more data to send.

- Window Size (16-bits): also known as flow control window. It is the amount of data octets from the one displayed in the acknowledgment field that this segment sender can accept (flow control).
- Checksum (16-bits): It holds the checksum value, which is calculated over the TCP header, TCP data and IP pseudo-header.
- Urgent Pointer (16-bits): If the URG flag is set, this field indicates the urgent pointer's current position as a positive offset from the sequence number in this segment. This is used to give priority to the data during processing.
- Options: It is used to provide additional features or parameters and can include Maximum Segment Size (MSS), Selective Acknowledgments (SACK), Timestamps, Window Scale Factor, etc.

c) Working Principle

The connection is initialized when the client sends a SYN segment with an initial sequence number (ISN_C). In response, the server sends SYN-ACK segment, acknowledging the clients SYN (ACK = ISN_C+1) and provides its own ISN_S. The client, in response to this, sends ACK segment to acknowledge server’s SYN (ACK = ISN_S + 1). The connection is established and known as Three-Way Handshake. The data is sent in segments, and each segment has its own sequence number. The receiver acknowledges each data by sending ACK. If no ACK arrives for a segment within a stipulated timeout period, the sender retransmits the segment. Duplicate ACKs can also cause faster retransmissions. When data is transmitted, there is a connection termination, which is a Four-Way Handshake. One side sends a FIN segment, and the other side responds with an ACK to this FIN. The other side, when it is ready sends its own FIN, and the first side then ACKs the FIN of the other side.

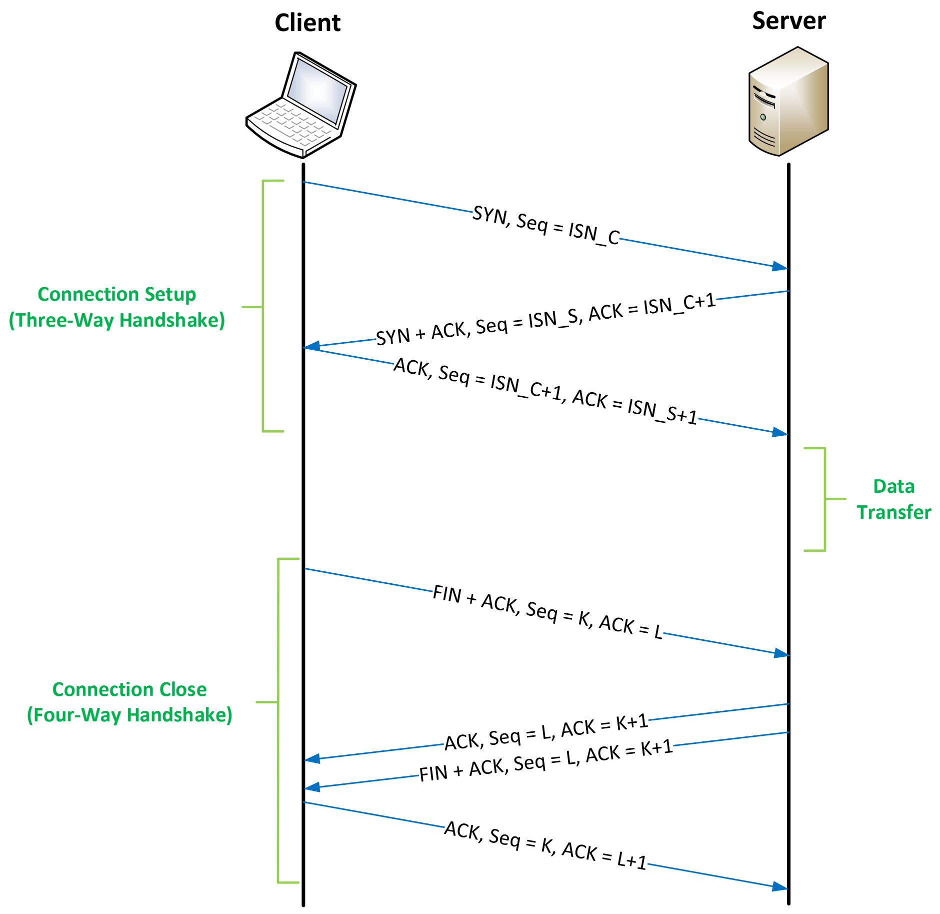

Client
Server
Connection Setup (Three-Way Handshake)
SYN, Seq = ISN_C
SYN + ACK, Seq = ISN_S, ACK = ISN_C+1
ACK, Seq = ISN_C+1, ACK = ISN_S+1
Data Transfer
Connection Close (Four-Way Handshake)
FIN + ACK, Seq = K, ACK = L
ACK, Seq = L, ACK = K+1
FIN + ACK, Seq = L, ACK = K+1
ACK, Seq = K, ACK = L+1

Figure 2-6: TCP Communication Flow [6]

d) *Advantages and Disadvantages*

Table 2-5: TCP Advantages vs. Disadvantages

Advantages	**Disadvantages**
Reliable connection as data delivery is guaranteed	Higher overhead and more connection management segments
Data is delivered in the correct order	Higher latency as it involves connection setup, sequence number processing, ACKs, retransmissions, and congestion control algorithms.
It has a flow control mechanism that prevents receiver buffer overrun	Retransmission and congestion control can lead to variations in packet arrival time, which is not good for real-time systems.
It has a congestion control mechanism that helps maintain overall network stability	More complexity to manage with respect to UDP connection

	If any segment is lost and the subsequent segments is received correctly, it may be held in a buffer by the receiver's TCP stack until the missing/lost is retransmitted and received. This leads to delay in delivery to the application

2.1.2 Serial Peripheral Interface (SPI)

Developed by Motorola in 1980s, SPI is a synchronous serial communication protocol used for short-distance communication, especially in embedded systems [7]. SPI has become a de facto standard widely adopted by most integrated circuit (IC) manufacturers for communication between microcontrollers and a variety of peripheral devices such as sensors, memory chips (Flash, EEPROM), analog-to-digital converters (ADCs), digital-to-analog converters (DACs), real-time clocks (RTCs), and other microcontrollers. Its ease of use, low hardware overhead, and high speed have also made it a staple in embedded system design. The architecture, operation, modes, advantages, and disadvantages of the SPI protocol will be discussed in this section.

2.1.2.1 Architecture

SPI employs a master-slave architecture, where the master device, which typically is a microcontroller. It initiates and controls the communication with one or more than one slave. The interface uses the following four logic signals [7].

- Serial Clock (SCLK): It is generated by the master to synchronize the data transmission. The data is shifted in or shifted out on the edges of the clock cycle.
- Master Out Slave In (MOSI): This line carries the data from the master to the slave.
- Master In Slave Out (MISO): This line carries the data from the slave to the master.
- Slave Select/Chip Select (SS/CS): This line is activated by the master to select the specific slave device. This signal is active low, and when a slave line is pulled low by this, it activates the SPI interface, and the slave starts responding to the SCLK and MOSI signal.

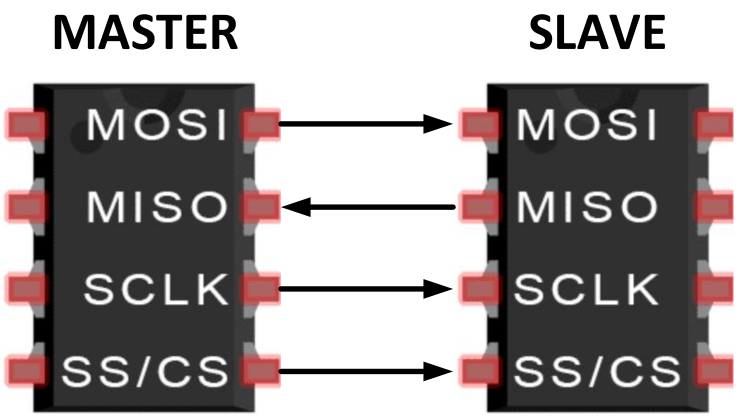


Figure 2-7: SPI Architecture [8]

The following are two architectural configurations of SPI:

a) Independent Slave Configuration

There is one clock, and MOSI and MISO are shared with all the slaves. Slaves are configured independently by the master, as all of them have an independent CS/SS line from the master. The master in this case has more than one CS/SS. It is the most commonly used architecture in SPI communication. The configuration is demonstrated in Figure 2-8: Independent Slave Configuration.

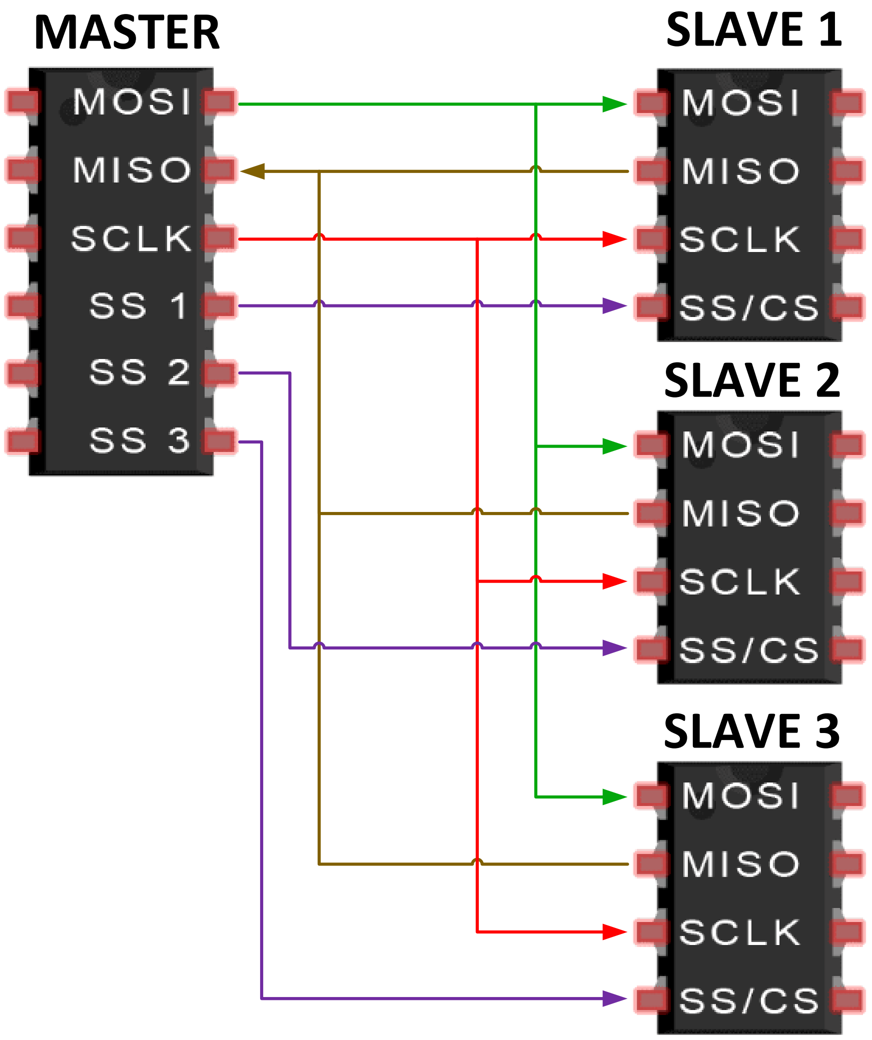


Figure 2-8: Independent Slave Configuration [8]

b) Daisy-Chain Configuration

There is only one connection for MOSI in this configuration. MISO of the first slave is connected to the MOSI of the second slave, MISO of the second slave is connected to the MOSI of the third slave and MISO of the last (third in this example) is connected to the MISO of the master. The advantage of this configuration is that the number of wires has been reduced. The configuration is demonstrated in Figure 2-9: Daisy-Chain Configuration.

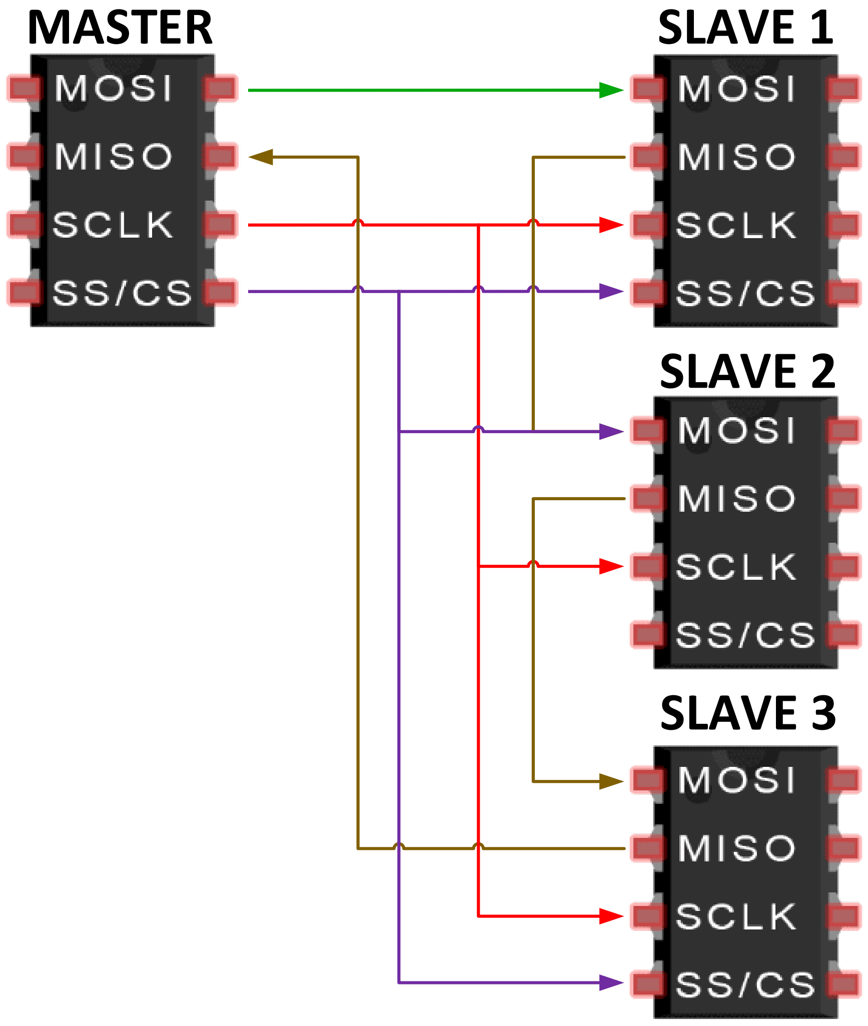


Figure 2-9: Daisy-Chain Configuration [8]

2.1.2.2 Data Transfer Operation

SPI communication is based on the synchronous shifting of data through shift registers in both the master and the selected slave. The master first selects a specific slave device by asserting its relative CS line (driving it low), and then it generates clock pulses on the SCLK line. On each clock cycle, the master shifts a bit out on the MOSI line to the slave's shift register. Meanwhile, the selected slave shifts a bit out on the MISO line to the master's shift register. This gives a full-duplex data exchange because the data is exchanged from master to slave and slave to master simultaneously. Conceptually, it is as though the contents of two shift registers (one in the master, one in the slave) are exchanged. The number of clock pulses, which subsequently is the bits exchanged per transfer, is typically 8 bits (one byte) but can be of other lengths (i.e., 16 bits, 32 bits, or even variable lengths) as defined by the specific slave device and configured in the master.

After the required number of bits has been transferred, the master de-asserts the CS line, resulting in the slave being deselected.

2.1.2.3 SPI Modes: Clock Polarity (CPOL) and Clock Phase (CPHA)

The timing relationship between the data bits and the clock signal is determined by two parameters, which are Clock Polarity (CPOL) and Clock Phase (CPHA). Both master and slave must be configured in the same mode in order for the communication to be effective. The two parameters give rise to four different modes of SPI, which are referred to as Mode 0, 1, 2, and 3.

- CPOL (Clock Polarity):
  - CPOL = 0: SCLK is low when idle (between transfers). The leading edge of the clock is a rising edge, and the trailing edge is a falling edge.
  - CPOL = 1: SCLK is high when idle. The leading edge of the clock is a falling edge, and the trailing edge is a rising edge.
- CPHA (Clock Phase):
  - CPHA = 0: The data is sampled on the first (leading) clock edge and is shifted out on the second (trailing) edge.
  - CPHA = 1: The data is shifted out on the first (leading) clock edge and is sampled on the second (trailing) edge.

The combination of these CPOL and CPHA provides four different SPI modes [9], which are:

- Mode 0: CPOL = 0, CPHA = 0
- Mode 1: CPOL = 0, CPHA = 1
- Mode 2: CPOL = 1, CPHA = 0
- Mode 3: CPOL = 1, CPHA = 1

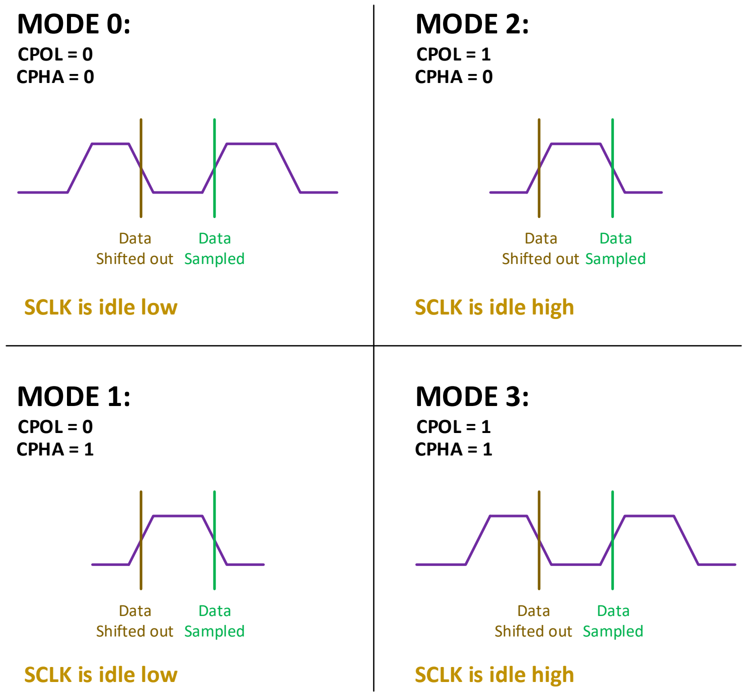


Figure 2-10: SPI Modes [9]

2.1.2.4 Advantages and limitations

SPI has several advantages that are the reasons for its popularity [8]:

- The protocol is simple with minimal software and hardware overhead. There are no complex state machines or addressing schemes within the data stream.
- SPI can run at very high data rates (tens of MHz, sometimes >100 MHz), but primarily limited by the maximum clock speed supported by the master and slave devices, and by PCB trace characteristics for signal integrity.
- Data can be sent and received simultaneously, hence ensuring Full-Duplex communication.
- Simple I/O structures, therefore, is power efficient compared to the other complex interfaces.
- Unlike I²C, the lines are actively driven; hence, they generally don't need pull-up resistor.
- Not restricted to 8-bit words thus allowing for flexible data frame sizes.

Despite its advantages, SPI has some limitations as well [8]:

- It requires more pins (at least four, plus one extra CS/SS if working with more than one slave). This can cause problems for microcontrollers with a limited number of I/O pins available.
- There is no concept of acknowledgement, as the master sends data to the slave without any particular acknowledgement from the slave. Once the master selects the slave, it assumes that the slave is present and working. Error check relies on the software check at higher levels.

- The selection of slaves is handled using dedicated CS lines. There is no address function within the SPI data stream.
- Only the master can generate the clock signal and initiate the communication; the slave cannot initiate transfer but can only hold the master via a separate “Slave Ready” line, which can be done in custom mode application.
- SPI is intended for short-distance communication, typically on the same PCB or across a cable of short length. Signal integrity degrades very rapidly over distance at high speed due to line capacitance, impedance mismatches, and noise.
- It has no built-in flow control. The master decides the speed, and the slave must be able to keep up to that pace.

2.2 Real-time Systems (Hardware-in-the-Loop)

The development and validation of complicated embedded systems, particularly considering the demanding automotive domain, rely heavily on the principles of real-time computing, like hardware-in-the-loop. Real-time systems ensure that the computations are performed within stringent time limits, which are a critical requirement for both safety and functionality in applications such as vehicle control. HIL simulation provides a robust framework for the verification of such real-time embedded components by creating a simulated platform that realistically mimics their operational conditions.

2.2.1 Real-time Systems: Core Concept and Characteristics

Real-time systems are such systems in which the correctness depends not only on the logical output but also on the time at which the output is produced [10].

2.2.1.1 Definition and Guiding Principles

"A real-time system is formally defined as one in which the correctness of the system depends not only on the logical result of computation but also on the time at which the results are produced." [11].

The primary guiding principle of real-time systems is that the operations are time-critical. It means that the validity of the operation’s result solely depends on the time at which it is delivered. It is not just about being fast but also being predictable with respect to the predefined time for the completion of the task. This has significant consequences with respect to the system design, focusing on the predictability, determinism, and worst-case execution time (WCET) analysis over average performance only. Key design concerns are the selection of scheduling methods, resource management policies, and system architectures that can commit to the timely response. Time criticality means that the system's correctness depends on meeting deadlines, responding to events quickly, and ensuring stability, especially in control applications that are common in automotive HIL simulations.

2.2.1.2 Classification of Real-Time Systems

The stringency of the timing requirements leads to the different classifications of real-time systems which are [12]:

- **Hard Real-Time Systems:** In hard real-time systems, failure to meet even a single deadline is considered a total system failure. These are often with catastrophic consequences. The system must ensure that all the critical tasks are completed within their assigned deadline under all the anticipated working conditions. Predictability and determinism are paramount in these systems. Examples of such systems are Flight Control systems, Anti-locking Braking systems (ABS) for automobiles, and industrial safety controllers.
- **Firm Real-Time Systems:** In firm real-time systems, missing a deadline also renders the result of a computation useless, but the system itself doesn't necessarily fail. While the delayed result is meaningless (or significantly less valuable), the system itself can continue to work. Occasional missed deadlines can be tolerated, but frequent misses will lead to a significant degradation in the quality of service or responsiveness of the system. Examples of such systems include certain types of sensor data acquisition where the old or previous data is irrelevant and a financial trading system where a late trade execution can be considered as a missed opportunity.
- **Soft Real-Time Systems:** In soft real-time systems, it is a desire to meet deadlines for optimal performance and for user satisfaction; however, missing them is not crucial to the system's core operation. The usefulness of the result decreases over time once its deadline has passed, but it may still have some value. The performance degrades gradually rather than leading to failure. Examples of such systems are multimedia streaming applications, where occasional frame drops can be tolerated, or some data logging systems where occasional delays in the data recording are acceptable.

2.2.1.3 Characteristics of Real-Time Systems

Real-time systems are defined by a set of characteristics that enable them to meet their timing-related obligations, which are

- **Determinism:** Determinism in a real-time system implies that given a set of inputs and an initial state, the system would produce the same outputs and follow the same sequence of state transitions, with operations taking a predictable amount of time. Temporal predictability is crucial in guaranteeing deadlines [13].
- **Predictability:** Predictability is the ability to establish, usually through analysis or formal methods before the deployment of the system, to check whether the system will consistently meet its specified deadlines for all anticipated

conditions, particularly the worst-case conditions. It involves bounding the execution times of the tasks and the behavior of system resources [14].

- **Timeliness (Low latency & Jitters):** Timeliness refers to the system's ability to complete operations within their stipulated deadlines. This is often characterized by:
  - Low Latency: minimizing the delay between the occurrence of an event, for example, sensor inputs, interrupts, etc., and the system's response to that.
  - Low Jitter: minimizing the variations in the latency for repeated instances of the same operation. Consistent timing is often as important as average speed [11].
- **Concurrency and Schedulability:** Real-time systems often manage multiple tasks running simultaneously, and each can have its own timing constraints.
  - Concurrency: The system must be designed to handle multiple activities or events that are running simultaneously.
  - Schedulability: It is the property of a set of tasks to be schedulable by an algorithm such that all of the tasks meet their requirements, i.e., deadlines. It is a formal process carried out to verify this [15].
- **Reliability and Fault Tolerance:** Reliability, usually being the important parameter in most of the systems, extends temporal aspects in real-time systems. The system must not only produce the correct logical output but also be able to do it in a certain time even in the presence of faults (within the specified limits). Fault tolerance strategies may include redundancy, error detection and correction, and recovery protocols that do not violate critical deadlines [16].

2.2.1.4 Real-Time Systems Implementation Challenges

Successful implementation of a real-time system involves navigating some well-known challenges that are inherent in both software and hardware design. It is necessary to overcome these challenges to achieve the desired reliability that characterizes real-time operation. Following are some key generic challenges:

- **Timing uncertainties Management:** There are some low-level system behaviors that can affect the precise timing of operations. The delay between a hardware interrupt signal and the execution of its interrupt service routine can introduce an event response [17]. Similarly, context switching overhead, which is the time consumed by the operating system to switch between the tasks, contributes towards non-application processing time that must be taken into account and be bounded.
- **Resource Contention and Synchronization:** During multitasking, tasks often share resources such as CPU, memory, I/O peripherals, etc. These uncontrolled accesses to these tasks can lead to race conditions or latency. Synchronization mechanisms such as semaphores and mutexes are necessary, but they can introduce their own complexities, which can potentially lead to priority inversion, meaning that the task with high priority is

forced to wait for a task with low priority that is currently using the resource, hence making it difficult higher-priority task to meet the deadlines.

- **Non-Determinism in Standard Platforms:** Many general-purpose computing platforms, which include operating systems, are not specifically designed for real-time determinism. Standard operating systems often employ schedulers that are optimized for average throughput that can lead to unpredictable task preemption times. Moreover, conventional network protocols like TCP/IP over standard Ethernet include mechanisms (e.g., TCP's congestion control and retransmissions) that introduce variable latencies, hence making them challenging for applications that require guaranteed timely data delivery [13].
- **Impact of Modern Processor Architectures:** Modern processors have advanced architectural features that can enhance the average performance but can cause complications to the predictability needed for real-time systems. Caches can alter the execution times based on miss/hit patterns, which leads towards difficulty in Worst-Case Execution Time (WCET) analysis. Similarly, instruction pipelines and branch prediction mechanisms can lead to mispredictions, introducing variability into task execution times [18].
- **Complexity of Timing Analysis and Verification:** Verifying that a system will meet all its deadlines under all possible conditions is a complex task. Worst-Case Execution Time (WCET) analysis of software components can be complex, especially for large codebases or for those with complex control flows. Moreover, testing and debugging real-time systems to identify and resolve timing-related faults (e.g., rare race conditions) requires specialized tools and techniques [19].

2.2.2 Hardware-in-the-Loop (HIL) Simulation: Principle and Applications

Hardware-in-the-Loop (HIL) simulation is an important technique for designing and testing complicated real-time embedded systems for the automotive, aerospace, and industrial automation industries. It allows comprehensive testing by connecting the physical hardware units with a simulated environment that emulates their operational environment strictly.

2.2.2.1 Definition and Primary Purpose

HIL simulation is a testing methodology in which an actual component of an embedded system, also known as the Device Under Test (DUT), is connected to a real-time simulator. The simulator executes mathematical models that can represent the behavior of the plant or other components with which the DUT is designed to interact [20].

The main purpose of HIL simulation is to enable automated and reproducible testing of embedded systems under realistic operating conditions. It is usually carried out early in the development phase, much before the possibility of full system integration. This enables early verification and validation of DUT’s performance and functionality.

2.2.2.2 Benefits of HIL Simulation in Automotive Development

The application of HIL simulation in the automotive industry has many advantages that enhance the development process:

- **Early Fault Detection:** The simulation allows us to identify and then rectify the design flaws in Electronic Control Units (ECU) and other associated components before they are physically integrated into a vehicle [21].
- **Testing under Extreme Conditions**: It allows safe testing of how a system responds in emergency situations, fault scenarios, or environmental extremes that would be highly risky to replicate if the testing were done with an actual vehicle.
- **Reduced Reliance on Physical Prototypes**: It decreases the dependency on expensive and limited vehicle prototypes, especially during the early stages of development.
- **Increased Test Coverage and Automation:** Helps in the execution of a wide range of tests, which also include regressive testing in an automated fashion, which leads to a thorough validation.
- **Accelerated Development Cycles:** It enables parallel testing and development of systems, thus allowing new automotive features or systems to arrive in the market at very short notice.

2.2.2.3 HIL Simulation Architecture

A standard HIL simulation setup comprises the following interconnected components [22]:

- Real-Time Target Computer: A high-performance platform for computations, i.e., dSPACE SCALEXIO, National Instruments PXI systems, which is responsible for executing the simulation models of the plant and the environment in real-time and also for managing the I/O operations of DUT.
- Plant Models: These are the mathematical representations of the physical systems (i.e., vehicle dynamics, sensor features, and actuator responses) with which the DUT interacts. They are typically created using modeling tools such as MATLAB/Simulink and then are compiled on a real-time target computer for execution.
- I/O Interface Hardware: These are the special-purpose hardware modules that are used to provide physical and electrical interfaces between the real-time target computer and DUT. It includes the capability to perform analog and digital signal acquisition/generation and interface for various communication buses such as Controller Area Network (CAN), Local Interconnect Network (LIN), FlexRay, Ethernet, and SPI. These interfaces ensure the simulator is able to send commands to the DUT and receive the desired response.
- Device under Test (DUT): an actual embedded system such as an automotive ECU, sensor, or actuator, that is to be tested. It receives the signal/command from the real-time target computer and then sends the output/response via I/O interface hardware.
- Host Workstation: A workstation/computer used for the overall management of the HIL system environment. The function of this computer is to develop plant models, configure

the real-time targets and I/O interfaces (for example, using tools like dSPACE ConfigurationDesk or NI VeriStand), design and execute the test scripts, monitor the execution in real-time (for example, using dSPACE ControlDesk or NI LabVIEW), and then perform post-test analysis and report.

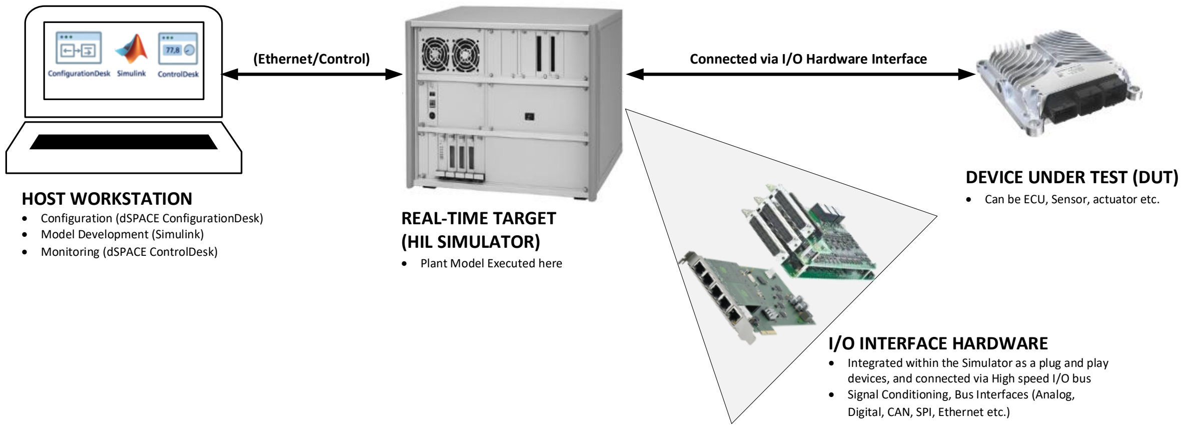

ConfigurationDesk
Simulink
ControlDesk
(Ethernet/Control)
Connected via I/O Hardware Interface
HOST WORKSTATION
• Configuration (dSPACE ConfigurationDesk)
• Model Development (Simulink)
• Monitoring (dSPACE ControlDesk)
REAL-TIME TARGET (HIL SIMULATOR)
• Plant Model Executed here
DEVICE UNDER TEST (DUT)
• Can be ECU, Sensor, actuator etc.
I/O INTERFACE HARDWARE
• Integrated within the Simulator as a plug and play devices, and connected via High speed I/O bus
• Signal Conditioning, Bus Interfaces (Analog, Digital, CAN, SPI, Ethernet etc.)

Figure 2-11: General Architecture of a Hardware-in-the-Loop (HIL) Simulation Setup [22]

2.2.3 Real-Time Constraints in HIL Systems

The validity of the HIL simulation is dependent on the real-time performance of the simulation itself. The interaction between DUT and the simulation plant must follow the strict time constraints to model the real-world behavior correctly.

2.2.3.1 Necessity of Real-Time operation in HIL

For accurate representation of HIL simulation as the system's operational environment, the plant models' execution and occurrence of I/O operations must be synchronized with the real-world characteristics of the DUT. In the closed-loop interaction where the simulation's output drives the DUT and the DUT's output influences the simulation, both must maintain temporal fidelity. Failure to operate in real-time can produce misleading results and instability in the simulated system [23].

2.2.3.2 Timing Accuracy in HIL for Automotive Systems

Automotive systems have tightly coupled closed loops, and the processes involved in them are very fast, thus imposing strict timing demands on the HIL simulations [20]:

- Accurate Simulation of Dynamics: Processes that involve power electronic switching, such as engine combustion events, occur at high frequency; therefore, they require that the model must be updated in correspondence to this high-frequency operation.
- Precise Signal Timing: The generation of the sensor/actuator signals sent to/from the DUT must take place with precise timing and low latency.
- Faithful Bus Communication: The simulation of automotive communication networks, i.e., CAN, LIN, FlexRay, etc., must accurately represent the protocol behaviors and timings in order to test the DUT's interfaces and functions correctly.

- Low Latency and Jitters: Data exchange between the simulator and the DUT should have low latency and minimum jitters to ensure that the DUT experiences conditions that represent its target application. Automotive sensors and actuators run at a very high speed; therefore, in order to simulate them accurately, these strict cycle times must be met [13].

2.3 Related Work

Several efforts at the academic and industrial levels have been made to explore the integration of embedded systems and real-time communication protocols using HIL platforms.

2.3.1 Hardware-in-the-loop testing for Embedded Systems

Numerous resources are available where embedded systems have been tested using HIL; however, the research conducted by Short and Pont in [24] can be used as a reference for testing an embedded system using HIL. They evaluated the embedded automotive controller under realistic conditions, using the real-time HIL simulation. Their setup simulated a motorway scenario and evaluated the performance of the Adaptive Cruise Control (ACC) system. The system was implemented using a CAN-based microcontroller. While their platform used the Infineon hardware and time-triggered scheduling for the communication, the core concept of their research aligns with the work conducted in the thesis, which is real-time interaction between physical embedded hardware and the simulated environment. Unlike the approach adopted by them, the work done in this thesis uses dSPACE SCALEXIO simulator for the generation of real-time signals and focuses primarily on the low-level communication validation of Ethernet and SPI using STMicroelectronics boards. There is a contrasting similarity in the goal, which is validating the timing and robustness of an embedded system communication under deterministic constraints.

2.3.2 Bridging Ethernet and SPI in Embedded Architecture

Yao Tong, in his research [25] designed an FPGA-based gateway that converts several asynchronous serial (UART) interfaces to Ethernet communication. The work done by him mainly focused on Unmanned Aerial Vehicles (UAV) and used VHDL-based state machines for managing multiple and an RTL8019AS network controller to frame and send packets. Although their system's primary focus was towards serial-to-Ethernet conversion, the overall concept was to bridge low-level communication protocols with Ethernet, which is closely aligned with the work done in the thesis. The current project creates a bridge that connects Ethernet to SPI using STMicroelectronics boards and the LwIP stack, providing a flexible, high-performing, and affordable solution for the intended Hardware-in-the-Loop (HIL) application. This software-based implementation on an MCU facilitates the real-time communication critical for HIL and embedded sensor emulation in automotive validation environments.

2.3.3 Real-Time Networking with STM32 and LwIP

Lightweight IP (LwIP) is a popular open-source TCP/IP stack for embedded systems where the use of a full-featured network stack is close to impossible because of limited resources. Zoican and Vochin, in their study [26], demonstrated successful use of LwIP on the Blackfin processors to implement an embedded client-server architecture for sensor networks, evaluating key performance parameters such as response time and buffer management. Their results obtained using Wireshark demonstrated that LwIP has acceptable latency and throughput even in the memory-limited real-time environments, which closely resemble the limitations encountered by the automotive embedded systems. Furthermore, another study conducted by Shang and Ding in their conference paper [27] presented the practical implementation of LwIP on the STM32F107 platform. They integrated Ethernet communication via RMII and Direct Memory Access (DMA) and showed interrupt-based packet handling with the use of Ethernet Interrupt Request (ETH_IRQ) and dedicated packet parsing routines. Additionally, their design also showed how LwIP can operate well in a non-RTOS setup by using the periodic polling method and STM32Cube peripherals, which is a design that directly relates to the communication framework used in this thesis. These studies validate the feasibility of using LwIP for real-time UDP communication in embedded systems, particularly for the applications that involve HIL interfacing. Building on this sound foundation, the present thesis employs LwIP on STMicroelectronics board to develop a specialized Ethernet-SPI bridge, which focuses on a stringent 1 ms cycle time and the custom framing protocol required for the emulation of a real-time automotive sensor.

2.3.4 Transport Protocols in Automotive Systems

Transport protocols play a significant role in determining the responsiveness and reliability of in-vehicle communication systems. Traditional automotive networks such as CAN and LIN are being replaced by Automotive Ethernet, which supports standardized IP-based communication stacks. TCP and UDP emerge as premier transport protocols in this context. Douss et al. [28] affirm that, although TCP allows for reliable communication, its complexity and latency make it less suitable for time-critical automotive applications. In contrast, UDP offers a lightweight, connectionless alternative, thus ideal for real-time applications such as Hardware-in-the-Loop (HIL) simulations. Kuo et al. [29] validated via OMNeT++ simulation that the performance of UDP is invariably superior to that of TCP in vehicular networks under real-time streaming conditions and offers better throughput and reduced delay. These findings align closely with the design choices made in this thesis, wherein UDP is used to interface with the STMicroelectronics board for fast, deterministic data transfer in a simulated automotive environment.

2.3.5 SPI Framing, CRC, and Data Integrity Approaches

When SPI is used in a demanding environment or when it is extended beyond its typical application, its data integrity and robust communication become major factors to be considered.

While SPI is a simple protocol and can operate at high speeds, it does not have a built-in capability of error detection. This point is also highlighted in studies like that of Karthick et al. [30], which emphasize incorporating error detection at the application level when the importance of data reliability is a must. To address this, Zitlaw et al. [31] conducted a comprehensive study to incorporate Cyclic Redundancy Codes (CRC) directly into SPI transactions. Various CRC polynomials and their trade-offs between error detection capabilities, computational overhead, and throughput impact were the main focus of their study. This led them to propose a specific Register-Transfer Level (RTL) architecture for CRC generation and verification. This work highlights the importance of incorporating error detection within SPI communication, which is also adopted in this thesis through the implementation of a software-based CRC-8 mechanism for validating SPI data packets.

Apart from error detection via CRCs, effective management of SPI data streams is benefited by using well-defined framing and synchronization strategies, especially in those cases that involve variable data types and require acknowledgments when bridged over a less reliable network like UDP. While describing SPI communication between MCUs, Lee et al. [32] emphasized the importance of synchronization methods, which included GPIO-based handshaking and checking of packet header information, in order to ensure data efficiency and prevent data loss. Their work pointed out the need for a mechanism that takes into account the readiness of the device and manages data integrity across transactions. The current thesis builds on similar principles by developing a custom framing protocol for all SPI communication between the STMicroelectronics boards. This protocol is designed to encapsulate variable-length SPI commands and data payloads (for example, like those emulating different types of sensors) and incorporates the software-based CRC for error checking. Furthermore, to enhance reliability over the UDP transport layer, concepts such as sequence identification and acknowledgment schemes, which are common in robust network protocol design, are considered within the application-level protocol of this thesis to manage data flow and detect packet loss.

2.4 Tools and Middleware

Several specialized tools and middleware platforms are integrated to develop and validate a real-time Ethernet-to-SPI communication framework for embedded automotive systems. All these tools are used as support in different development stages, such as system design, low-level firmware development, protocol stack configuration, model-based simulation, and HIL testing. An overview of these tools and middleware platforms that form the foundation for such development efforts are provided below.

2.4.1 Microcontroller Development Ecosystem: STMicroelectronics STM32

The STMicroelectronics STM32 family is based on ARM Cortex-M cores, and it offers a versatile platform for embedded applications [33]. It provides a balance of processing power and a set of integrated peripherals, including Ethernet MAC, SPI, DMA, and general-purpose I/O [34]. STMicroelectronics STM32 boards are commonly developed using Integrated Development Environments (IDEs), such as STM32CubeIDE. These IDEs facilitate the development, compilation, and debugging of C/C++ firmware. For simplification, graphical tools like STM32CubeMX are used for the generation of low-level initialization code and configuration of peripherals (like SPI, Ethernet, DMA, timers, GPIOs). Software interaction with the MCU hardware is managed via STMicroelectronics’ Hardware Abstraction Layer (HAL) drivers, which provide a standardized API for peripheral control. Programming and in-circuit debugging are mostly facilitated by hardware tools like the ST-LINK, but in some STMicroelectronics boards there are built-in debuggers used for it.

2.4.2 Embedded Networking Middleware: LwIP

LwIP is a well-known open-source TCP/IP (and UDP/IP) stack designed for embedded systems where memory and processing resources are limited [35]. Because of its small footprint, which provides support for network protocols such as UDP, and its portability, which makes it suitable for bare-metal (non-RTOS) operation through periodic polling mechanisms, it is a common strategy used for integration. STMicroelectronics includes LwIP within its CubeMiddleware packages, which simplifies its deployment on STM32 Microcontroller boards that have integrated Ethernet capabilities. They provide networking functionalities (IP, UDP, TCP, etc.) and are ported to the microcontrollers and the operating environment.

2.4.3 Prototyping, Analysis and Verification Tools

Developing and validating embedded systems often involves the use of a variety of auxiliary tools. These tools play a crucial role in prototype designing and detailed analysis. Python is a high-level scripting language that has extensive libraries for network socket programming which makes it helpful in prototype designing and testing. Python is employed to simulate communication endpoints and perform preliminary data processing. Python scripts can be executed within command-line environments such as Windows PowerShell. For verification of the results produced, network protocol analyzers, such as the widely used open-source tool Wireshark, can be used for capturing the packets and analyzing the network traffic. Their capability to decode the protocols and display packet information is very helpful for debugging purposes, packet arrival time, and data integrity.

2.4.4 Hardware-in-the-Loop (HIL) Simulation Environments

HIL simulation is an important part of the testing of control systems, especially in the automotive industry. This environment typically consists of:

- **Configuration Management Tool:** Software such as dSPACE ConfigurationDesk is used to configure the HIL setup; it maps the signals between the simulator and the DUT. It also manages different experimental configurations.
- **Model-Based Design Software:** MATLAB/Simulink is widely used for creating, simulating, and verifying system models. These are integrated with HIL platform through special toolchains (e.g., dSPACE Real-Time Interface - RTI), which enable automatic code generation for real-time hardware.
- **Experimental and Control Software:** dSPACE ControlDesk is a GUI-based software that allows you to manage HIL experiments, visualize the real-time data, and interact with the simulation. Moreover, it is also used to automate test procedures.
- **Real-Time Simulation Hardware:** It is a powerful computing platform designed for executing mathematical models of the physical plant with deterministic timings. dSPACE SCALEXIO is one of the examples of the hardware used as a real-time simulator.

Chapter 3

System Architecture

3.1 Hardware Overview

The hardware architecture of the system plays a vital role in enabling real-time communication, processing, and simulation for automotive testing applications. The work done in this thesis revolves around the integration of embedded microcontroller platforms with a real-time HIL simulation environment that is supported by high-speed communication Ethernet and SPI interfaces. Selection of all the hardware components is done considering their ability to meet the system's requirements for low latency and reliable data exchange. An overview of key hardware elements used in the implementation of this project is described in this section, which includes the STM32 microcontroller boards, communication interfaces, and the dSPACE processing unit within the SCALEXIO simulation platform.

3.1.1 dSPACE HIL Simulation Platform

HIL simulation functionality of this project was conducted using a dSPACE real-time simulation platform that is a widely recognized platform in the automotive industry for testing and validating embedded control systems. This platform provides the real-time processing power and I/O capabilities that are required to execute complex vehicle and environment models and interface with physical ECUs or, as in this thesis, custom embedded hardware. dSPACE associated hardware specifically used for this project is described as follows.

3.1.1.1 dSPACE SCALEXIO LabBox Real-Time Chassis

The HIL hardware platform utilized in this project is the dSPACE SCALEXIO LabBox. The SCALEXIO LabBox is a modular system that is specifically designed for laboratory use and provides a robust environment for HIL simulation and Rapid Control Prototyping (RCP) tasks

[36]. Specifically, an 8-slot version of the LabBox was employed that offers sufficient capacity to house the necessary processing and I/O hardware. It serves as the central chassis, providing power, cooling, and the high-speed I/O carrier network (IOCNET) for connecting the installed dSPACE boards, thereby forming a real-time simulation unit [36].

Figure 3-1: dSPACE SCALEXIO LabBox (8-slot version) [36]

3.1.1.2 dSPACE DS6001 Processor Board

Real-time processing capability within the SCALEXIO chassis was provided by the dSPACE DS6001 Processor Board. It is a high-performance, two-slot board that is designed for executing the real-time applications compiled from Simulink models [36]. The DS6001 processing unit is an Intel® Core™ I7-6820EQ multi-core processor used for running the simulation models. It also has an ARM® Cortex®-A9 dual-core coprocessor that is dedicated to manage the communication with the host PC [36].

Figure 3-2: DS6001 Processor Board [36]

With respect to this project, the DS6001 board is equipped with multiple Gigabit Ethernet interfaces. One Ethernet port is typically utilized for connection to the host PC to facilitate the model deployment, experiment control via dSPACE ControlDesk, and log data. A second dedicated Ethernet port, referred to as the "Ethernet Adapter 1 channel type," is available for connecting Ethernet-based I/O for direct communication with the DUT or an interface board like the STM32 Master board (as in this case) [36]. This second port, which supports 10BASE-T, 100BASE-TX, and 1000BASE-T Ethernet standards, was configured to establish the UDP communication link with the STM32 Master board for transmitting real-time commands and receiving data. The system operates with a Linux-based real-time operating system on the DS6001 [36].

Table 3-1: Technical Specification of DS6001 Processing Unit [36]

Specification	**Description**
Processor Architecture	Intel® x86, multi-core real-time capable
Real-Time OS	dSPACE RTOS (Proprietary, deterministic)
Memory	16 GB DDR4 RAM (typical), with fast memory access
Ethernet Interfaces	Up to 4× Gigabit Ethernet ports (UDP/TCP)
I/O Connectivity	Compatible with I/O units via PXI or SFP extensions
Model Execution	Supports MATLAB/Simulink auto-code generation (RTI)
Synchronization Support	IEEE 1588 PTP, dSPACE clock modules

3.1.2 Embedded Microcontroller Platform

The core logic for the Ethernet-to-SPI bridging and the functionalities to emulate sensor is implemented using a pair of STMicroelectronics STM32F207ZG microcontrollers. These MCUs are hosted on Nucleo-144 development boards (e.g., NUCLEO-F207ZG [37]). It was selected for its Arm® Cortex®-M3 core operating up to 120 MHz, sufficient Flash and SRAM (up to 1 MB Flash, 128+4KB SRAM), and a comprehensive set of integrated peripherals that are crucial for this project, including an Ethernet MAC, multiple SPI interfaces, and DMA controllers [38]. The two boards are configured in a master-slave topology to implement the communication bridge. The key roles and peripheral utilization for each board are detailed in the table below.

Table 3-2: Hardware Role and Peripheral Utilization of STM32F207ZG Boards

Feature	Master STM32F207ZG Board	Slave STM32F207ZG Board
Primary Role	Ethernet-SPI Gateway: Receives UDP commands (from dSPACE or Python), initiates SPI to Slave, and sends UDP responses/echoes (to Python/Simulator).	Sensor Emulator: Acts as SPI slave, processes commands from Master, returns emulated sensor data (Speed, IMU) over SPI.
SPI Interface	Configured as SPI Master – Full Duplex	Configured as SPI Slave – Full Duplex
Ethernet MAC	Utilized for UDP/IP communication with dSPACE HIL and Python host.	Not used in case of Slave board
DMA Controller(s)	Used for efficient full-duplex SPI data transfers (transmit and receive).	Used for efficient full-duplex SPI data transfers (transmit and receive), configured in circular mode for continuous data handling.
GPIO(s)	Software-controlled Slave Select (NSS) output for SPI. PG4 used as GPIO output for Slave selection purpose	Hardware NSS input for SPI slave selection.
CRC Calculation	Software-implemented CRC-8 for custom SPI frame integrity.	Software-implemented CRC-8 for custom SPI frame integrity.
Operating Environment	Bare-metal (non-RTOS), main loop with event-driven logic (UDP reception triggering SPI)	Bare-metal (non-RTOS), main loop with event-driven logic (SPI peripheral or EXTI for NSS assertion, triggering DMA and processing).

3.1.2.1 Master Board (STM32F207ZG)

The master STM32 board plays the most important role, which is to bridge the Ethernet network with the SPI-based slave device. It receives real-time commands encapsulated in UDP packets from the dSPACE HIL simulator or a Python test script (prototype) via its integrated Ethernet MAC. The master board upon receiving a UDP command, parses the command and initiates the full-duplex SPI transaction to the slave board. This involves constructing a custom-created SPI frame that includes the command and a software-based CRC-8 checksum. SPI communication is managed using one of the STM32F207ZG's SPI peripherals (SPI1) that is configured in master mode, with DMA channels assigned for both transmitting and receiving data to minimize CPU load and to ensure timely data handling [34]. A general-purpose I/O pin (PG4) is used as GPIO output to implement a software-controlled Slave Select (NSS) signal for the SPI interface. After the SPI transaction with the slave is complete, the master board processes the received SPI response (e.g., echoed data or sensor values), verifies its integrity using CRC-8, and then, if required by the communication sequence, formulates and transmits a UDP packet back to the simulator or the Python host.

3.1.2.2 Slave Board (STM32F207ZG)

The slave STM32 board is used to emulate the behavior of an automotive sensor. It operates as a full-duplex SPI slave and its SPI1 peripheral is configured to respond to data received from the master board. The slave is configured in circular mode, with DMA channels assigned for both reception and transmission of the data. It handles the SPI data coming from the master board and then loads its transmit buffer with an appropriate response. After receiving a valid framed command from the master (after CRC-8 verification), the slave's application logic interprets the command. Based on the received command, it either prepares an acknowledgment or non-acknowledgment (ACK/NACK) frame or generates emulated sensor data (such as vehicle speed, IMU gyroscope, or accelerometer values, as defined by the received sub-command). The response is also encapsulated in the custom SPI frame format with a CRC-8 checksum and is then made available in its SPI transmit buffer for the master to retrieve during a subsequent SPI transaction. The hardware NSS input pin of the slave's SPI peripheral is driven by the master board to enable communication.

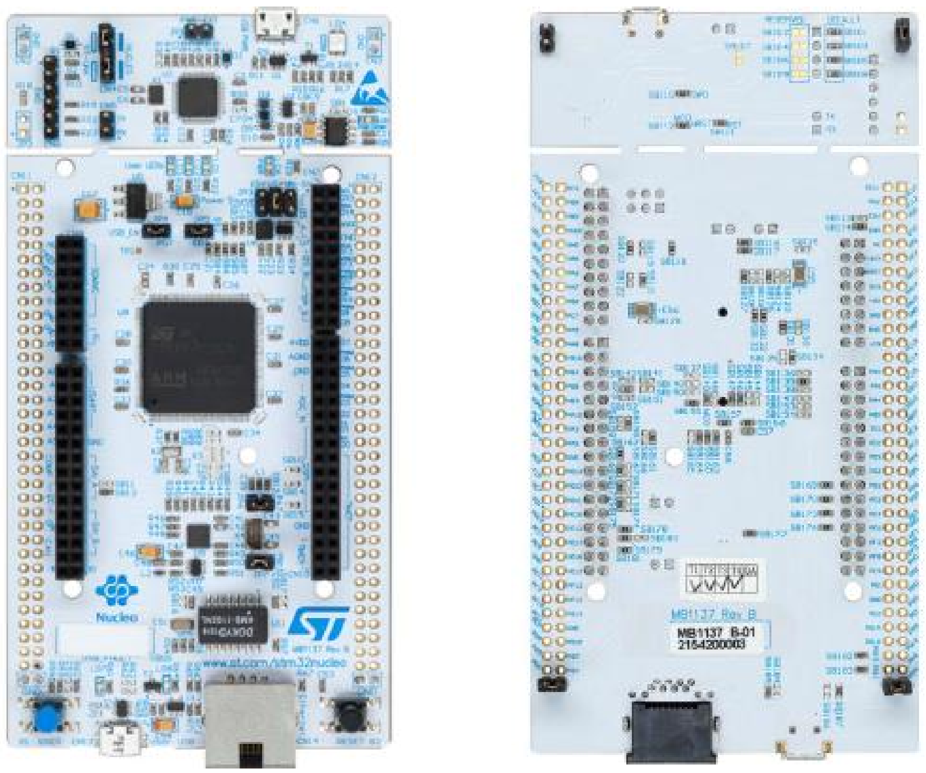


Figure 3-3: Top and Bottom view of STM32F207ZG Board [37]

3.1.3 Host Workstation and Prototyping PC

3.1.3.1 dSPACE Host Workstation

This PC is used for dSPACE software; MATLAB/Simulink for model design, dSPACE ConfigurationDesk for HIL setup, and dSPACE ControlDesk for experiment execution and monitoring. It is connected to the dSPACE SCALEXIO DS6001 processor board via Ethernet for deployment of the model and managing the HIL system.

3.1.3.2 Prototyping and Test PC (Python Environment)

This PC is used as a part of prototype testing. Python script is running on this PC and is used for initial system testing, sending UDP commands to STM32 master, and receiving/displaying echo responses. This PC is connected to the same network as the STM32 master.

3.1.4 Physical Interconnects

Proper operation of the developed systems heavily relies on the correct physical interconnections between its constituent hardware components. Details of these critical electrical and communication links are depicted below.

3.1.4.1 Ethernet Network Connectivity

The dSPACE DS6001 Processor board, the Master STM32F207ZG Nucleo board, and the PC used for prototype testing are interconnected via a standard 100/1000 Mbps Ethernet network. The Master STM32F207ZG Nucleo board is connected directly to the prototype PC via Ethernet cable (CAT5/6) for testing and command/response exchange. Similarly, for HIL setup, the Master STM32 board was connected directly to the Ethernet port on the dSPACE DS6001 Processor Board enclosed in the SCALEXIO chassis. Static IP addresses are configured on all these devices to ensure that the network paths between them for the UDP-based data exchange is reliable.

3.1.4.2 SPI Interfaces (Master STM32 to Slave STM32)

SPI link is established between the Master and Slave STM32F207ZG Nucleo board using their respective SPI1 peripheral for full-duplex communication. The specific pin connections are as follows:

- **SCK:** Master STM32 pin PA5 (Nucleo CN7, D13/SPI_A_SCK) is connected to the Slave SMT32 pin PA5 (Nucleo CN7, D13/SPI_A_SCK).
- **MISO:** Master STM32 pin PA6 (Nucleo CN7, D12 / SPI_A_MISO) is connected to the Slave SMT32 pin PA6 (Nucleo CN7, D12 / SPI_A_MISO).
- **MOSI:** Master STM32 Pin PB5 (Nucleo CN7, D22 / SPI_B_MOSI, an alternate function for SPI1_MOSI) is connected to Slave STM32 Pin PB5 (Nucleo CN7, D22 / SPI_B_MOSI, configured as SPI1_MOSI input). This alternate MOSI pin (PB5) was selected on the Master board to avoid conflict with by default MOSI pin PA7 which is shared by Ethernet peripheral as well.

- **NSS:** The Slave select mechanism is implemented using a software-controlled approach. A GPIO pin, PG4, on the Master STM32 board is connected to the hardware NSS input pin PA4 (Nucleo CN7, D24 / SPI_B_NSS) of Slave STM32 board. The Master firmware manages the assertion (active low) and de-assertion of its PG4 pin to enable and disable the SPI slave for each transaction.
- Ground (GND) pins on both boards were also connected to ensure a common reference.

3.1.4.3 Power Supply

Both STM32F207ZG development boards (Master and Slave) were powered independently via their respective USB ST-LINK connections. They are provided power using the standard USB ports on personal laptops. The dSPACE SCALEXIO system and its DS6001 Processor Board were powered by their dedicated mains supply.

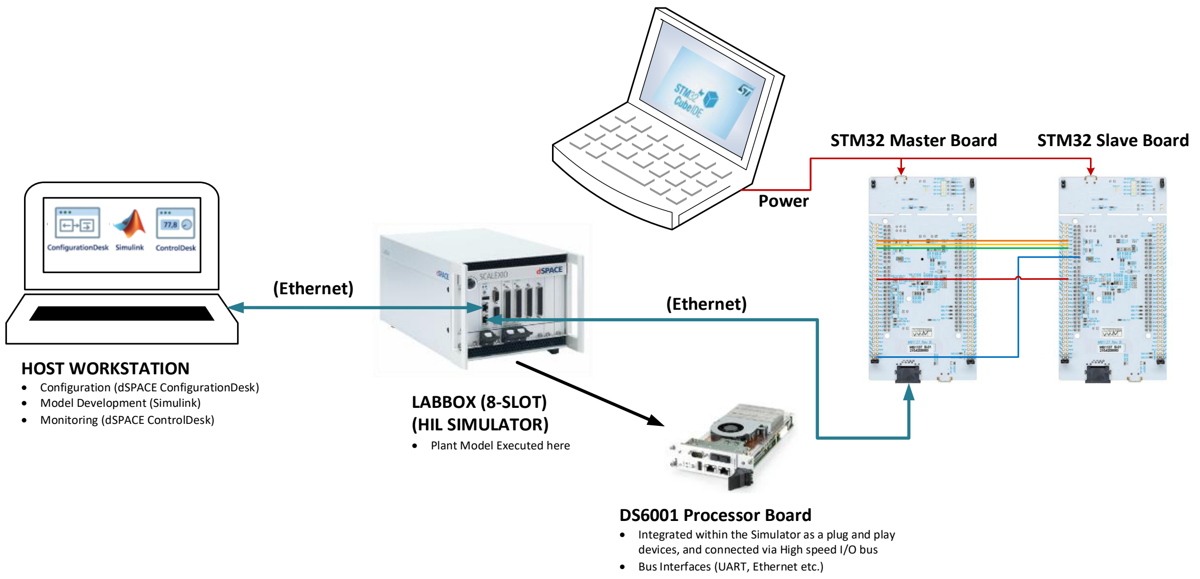

STM32 Master Board
STM32 Slave Board
Power
ConfigurationDesk
Simulink
ControlDesk
(Ethernet)
(Ethernet)
HOST WORKSTATION
• Configuration (dSPACE ConfigurationDesk)
• Model Development (Simulink)
• Monitoring (dSPACE ControlDesk)
LABBOX (8-SLOT)
(HIL SIMULATOR)
• Plant Model Executed here
DS6001 Processor Board
• Integrated within the Simulator as a plug and play devices, and connected via High speed I/O bus
• Bus Interfaces (UART, Ethernet etc.)

Figure 3-4: Overall System Hardware Architecture

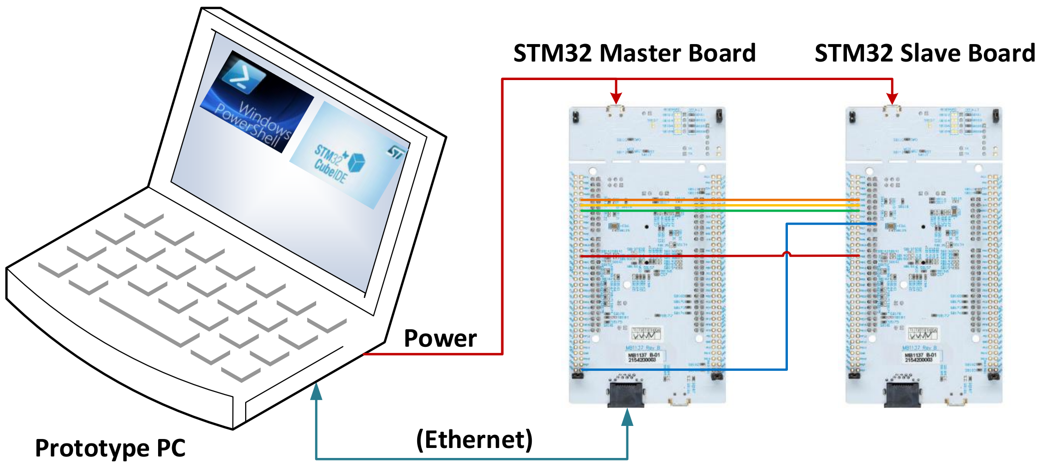


Figure 3-5: Prototype Hardware Architecture

3.2 Software Stack

The functionality of the Ethernet-SPI bridging and the sensor emulation is carried out through firmware developed for the STMicroelectronics STM32F207ZG microcontrollers. This firmware manages all the peripheral interactions, network communication via LwIP, and the customized application layer protocol designed for real-time data exchange. A bare-metal (non-RTOS) approach was used for both the Master and Slave boards, which allows control of the hardware directly and minimizes the system overhead, thereby supporting the timing requirement for HIL simulation. Detailed hardware architectures of each board are presented as follows.

3.2.1 Master STM32 Board: Software Architecture

Master STM32 board's firmware is designed to function as the central gateway. It manages the bi-directional communication, receives UDP commands over Ethernet from the external environment (dSPACE HIL simulator or Python-based test scripts running over PC) and then communicates with Slave STM32 board via full-duplex SPI protocol.

3.2.1.1 Operating Environment and Core Initialization

The Master board is operated using a bare-metal approach, with system execution being managed through the main super-loop. Essential low-level initialization, which comprises system clock configuration, GPIO setup (including pins for software-controlled SPI NSS), DMA channels, the SPI1 peripheral (configured as master full-duplex), and the Ethernet MAC, is performed using STM32CubeMX-generated code and the STM32 HAL drivers. LwIP network stack is also initialized at the beginning to prepare for the network operations. Inside the main loop, periodic calls to *LwIP's ethernetif_input()* and *sys_check_timeouts()* ensure that the Ethernet interface is polled for incoming UDP packets and that LwIP's internal timing requirements are met. The main loop also includes the logic used for managing the state of SPI communication and handling potential transaction timeouts.

3.2.1.2 LwIP Stack Integration and UDP Communication

The LwIP stack is important for the Master board's functionality as it provides UDP/IP services over the STM32F207ZG's Ethernet MAC. A UDP Protocol Control Block (PCB) is established during initialization *(udp_server_init_master)* and is bound to listen to a predefined static IP address and UDP port (e.g., 50000) for incoming datagrams.

A dedicated UDP receive callback function *(udp_receive_callback)* is listed with LwIP. When a UDP packet arrives on the designated port, this callback function is called by the LwIP stack. The callback function is responsible for parsing the payload, which may contain either raw data (such as floating-point speed values) or control commands (such as a 1-byte sub-command that indicates the type of data stream that the Slave should provide, e.g., idle, speed, IMU's gyroscope, or IMU's accelerometer data). The IP address and port of the UDP sender are read using *(lastSenderAddr,* *lastSenderPort)* to enable subsequent transmission of responses or echoed data. Based on the UDP command and the current operational state of SPI (e.g., ensuring the SPI interface is currently idle via *current_spi_state* or *awaitingAck* flags), the Master prepares and initiates an SPI transaction to the Slave board.

Based on the data received from the Slave, which can be sensor data or the acknowledgment/non-acknowledgment (ACK/NACK) messages, the Master's firmware constructs a response as an appropriate UDP payload. This payload at times also includes the data type identifier from the Slave's SPI response to facilitate the host so that it can make proper interpretation. LwIP's *pbuf_alloc()* and *udp_sendto()* functions are then used to send these datagrams to the sender IP address and port.

3.2.1.3 SPI Master Control and Protocol Management

The Master board's SPI1 peripheral is configured as master full-duplex and uses DMA (*HAL_SPI_TransmitReceive_DMA*) for non-blocking data transfer. The essential part of the SPI control logic is to manage the custom communication protocol that is designed for flexibility and robustness.

This custom protocol comprises a defined frame structure *(build_frame function)* used for all SPI communication. Each frame includes a *START_BYTE* delimiter (e.g., 0xAA), a Length field (defines the length of the subsequent Type and Payload fields), a Type field (e.g., *FRAME_TYPE_MASTER_DATA* to send data/commands to the slave, or *FRAME_TYPE_CMD_CONTROL_SLAVE* for specific slave mode control), a variable-length Payload (which can contain actual data like speed values, or sub-commands for the slave, and might include a sequence ID like txSeqID), a software-calculated 8-bit Cyclic Redundancy Check (CRC-8 is generated by using crc8_calculate covering essential frame parts), and an *END_BYTE* delimiter (e.g., 0x55).

SPI transactions is initiated based on the incoming UDP commands. Software-controlled Slave Select (NSS) that uses a dedicated GPIO pin (e.g., PG4), is asserted before each SPI DMA transfer and de-asserted upon completion or error. The function *HAL_SPI_TxRxCpltCallback* is called when a full SPI DMA transaction (both transmit and receive) completes. Inside this callback, the Master de-asserts NSS and analyzes the received SPI frame (rxFrame) from the Slave. During this analysis, it validates the frame structure (Start/End Bytes), verifies the CRC-8 checksum, and checks that whether the received frame type and sequence ID are as per the expected values or not. Depending on the received frame type (e.g., *FRAME_TYPE_SLAVE_ECHO*, an ACK like 0x04, or a NACK like 0x05), the Master either confirms successful configuration/data exchange *(configAcked = true)*, notes a negative acknowledgment *(configAcked = false)*, or prepares the received sensor data payload for echo transmission back to the UDP host.

To ensure proper communication, the master implements time detection for SPI transactions. If an SPI completion (signaled by *spiReady* flag and *HAL_SPI_TxRxCpltCallback*) does not occur within the predefined time *SPI_TIMEOUT_MS* which is tracked via *spiTxStartTick* and

HAL_GetTick(), a timeout is declared. Retry logic *(check_timeout_and_retry, configRetryCounter, MAX_RETRIES)* is employed, in which the Master retries to send the *CONFIG* command or the last data command a few times before resetting its state or flagging an error. SPI hardware errors are detected by the HAL and are handled in the *HAL_SPI_ErrorCallback*, which typically aborts the current SPI operation, re-initializes the SPI peripheral, and resets the relevant state flags to allow for subsequent recovery attempts.

3.2.2 Slave STM32 Board: Software Architecture

Slave STM32 board's firmware is designed to function as a responsive sensor emulator that reacts to the SPI commands received from the Master board and transmits the emulated data back as response to it. The design provides efficient, interrupt driven processing and continuous data availability through the use of circular DMA for SPI communication.

3.2.2.1 Operating Environment and Core Initialization

The Slave board is operated using a bare-metal approach, with system execution being managed through the main loop and the interrupt service routines. Essential low-level initialization, which comprises system clock configuration, Hardware SPI NSS, DMA channels, the SPI1 peripheral (configured as slave full-duplex), is performed using STM32CubeMX-generated code and the STM32 Hardware Abstraction Layer (HAL) drivers.

The initialization involves configuring SPI1 peripheral in slave mode with hardware NSS input enabled. Doing so, SPI peripheral can be automatically enabled/disabled by the NSS signal provided by the Master board's GPIO. The DMA controller is configured with two streams dedicated to SPI1, one for reception (*SPI1_RX*) and one for transmission (*SPI1_TX*). Both DMA streams are programmed to operate in circular mode, which allows continuous data transfer between the SPI1 Data Register (*SPI1_DR*) and circular memory buffers (e.g., *spi_rx_dma_circular_buf* and *spi_tx_dma_circular_buf*) that are pre-allocated in the code. The size of these buffers is determined by *MAX_FRAME_SIZE* and *NUM_BUFFERS_IN_CIRCULAR* to support a ping-pong buffering scheme.

To ensure correct operation of the circular DMA with distinct processing for each half of the buffer, specific DMA control register bits (such as *CIRC* for circular mode, *MINC* for memory increment, and *HTIE/TCIE* for half-transfer and transfer-complete interrupt enable in the *DMA_SxCR* register) are configured post-CubeMX generation via HAL functions or direct register access. The transmit circular buffer (*spi_tx_dma_circular_buf*) is pre-filled with a default valid SPI frame (e.g., an ACK frame or idle sensor data) during initialization. This ensures that the Slave always has data ready for transmission when the Master initiates an SPI transaction by asserting NSS and providing the clock. Following peripheral setup, the continuous *HAL_SPI_TransmitReceive_DMA()* in circular mode is initiated.

3.2.2.2 Interrupt-Driven SPI Data Processing via DMA Callbacks

Data processing logic on the Slave board is event-driven, that is triggered by DMA interrupts when a new data from the Master is received. The STM32 HAL provides weak-aliased callback functions for DMA events and are implemented in the Slave's firmware [39]:

- ***HAL_SPI_RxHalfCpltCallback(SPI_HandleTypeDef *hspi)***: This callback is called when the SPI1_RX DMA stream has filled the first half of its circular buffer. The firmware within this callback is able to access the first segment of the received data.
- ***HAL_SPI_RxCpltCallback(SPI_HandleTypeDef *hspi)***: This callback is called when the SPI1_RX DMA stream has filled the second half of its circular buffer (this means that it has completed a full circular pass for that buffer from the DMA's perspective before it wraps around). The firmware within this callback is able to access the second segment of received data.

A processing function (e.g., *process_received_data_and_prepare_tx* or *handle_frame*) is called within both these RX callbacks, and this function is responsible for analyzing the received SPI frame (validating start/end bytes, length, and CRC-8), interpreting the Master's command (which includes any sub-commands for sensor data type selection), and then preparing an appropriate response. The frame that is generated as a response is then placed into the corresponding half of the SPI TX circular DMA buffer, and then it is made ready for the DMA to transmit when the Master next clocks data from the Slave. This mechanism ensures that while one buffer half is being processed by the CPU, the other half is available for the DMA, ensuring continuous data flow.

TX DMA completion callbacks (*HAL_SPI_TxHalfCpltCallback*, *HAL_SPI_TxCpltCallback*) are also defined by the HAL, and their role in this architecture is to signal that a transmit buffer segment has been sent [39]. The data itself is prepared in response to the RX events. The *HAL_SPI_ErrorCallback* handles SPI hardware errors detected by HAL [39], and it typically attempts to re-initialize the SPI and DMA peripherals and resets the internal state variables to ensure communication can recover.

3.2.2.3 Custom SPI Frame Protocol and Sensor Emulation Logic

The Slave board strictly follows the custom SPI frame protocol created for communication with the Master board. This includes a START_BYTE, Length, Type, variable Payload, software-calculated CRC-8, and an END_BYTE.

The Type field within a frame received from the Master controls the Slave's action. If the type indicates a control command from the Master (e.g., *FRAME_TYPE_CMD_CONTROL_SLAVE* with a specific sub-command in the payload), the Slave updates its internal state to decide which type of sensor data to emulate (e.g., *SUB_CMD_SET_STREAM_IDLE, SPEED, GYRO, ACCEL*). Slave then prepares a frame (*FRAMETYPE_SLAVE_ACK*), including the echoed sub-command and a status byte as its response.

If the Master initiates an SPI transaction to poll for data (by sending a generic command or a previously set control command again), the Slave, generates appropriate emulated sensor data based on its current internal state (*current_data_stream_type_to_send*). The sensor data are as follows:

- Vehicle Speed: A floating-point value, framed as *FRAME_TYPE_DATA_VEHICLE_SPEED*.
- IMU Gyroscope: Typically three 16-bit integer values, framed as *FRAME_TYPE_DATA_IMU_GYRO*.
- IMU Accelerometer: Typically three floating-point values, framed as *FRAME_TYPE_DATA_IMU_ACCEL*.

Each response frame sent by the Slave also includes a sequence ID (e.g., *echoCounter* or by echoing the Master's sequence ID if it is present in the command frame) and is protected by a CRC-8 checksum. If an unrecognized or corrupt frame is received, the Slave prepares a *FRAME_TYPE_SLAVE_NACK* response. This entire response generation and framing process occurs within the logic called from the DMA RX callbacks, hence ensuring that the transmit buffer segment is ready before the Master clocks it out. A timeout mechanism (*lastRxTick* and *SPI_TIMEOUT_MS*) is also implemented; if no SPI activity (DMA callbacks) occurs for a defined period of time, the SPI and DMA are considered stalled and are re-initialized to a known state.

3.2.3 Data Structures and Communication Flow

Data structures and coordinated communication sequences define the reliable and efficient exchange of information between the HIL environment, the Master STM32 board, and Slave STM32 board. These elements ensure that the commands are correctly interpreted and the sensor data is accurately framed.

3.2.3.1 Core Data Structures for SPI Communication

Two primary data structures support the inter-microcontroller communication logic: the customized SPI frame format used for all master-slave exchanges and the internal C-language structures utilized to represent the sensor data payloads prior to framing or after parsing.

Customized SPI frame structure depicted in Figure 3-6: Custom SPI Frame Structure. This frame is managed within uint8_t array buffers and is designed with distinct fields for clarity, error detection, and support for variable payloads. Each frame starts with a *START_BYTE* (e.g., 0xAA) as a preamble, followed by a *LENGTH* field (1 byte) that specifies the combined length of the subsequent *TYPE* and *PAYLOAD* fields. The *TYPE* field (1 byte) is crucial for distinguishing the frame's purpose, with different codes for Master-to-Slave commands (e.g., *FRAME_TYPE_CMD_CONTROL_SLAVE*), Slave-to-Master acknowledgments (e.g., *FRAME_TYPE_SLAVE_ACK, FRAME_TYPE_SLAVE_NACK*), or Slave-to-Master data transmissions (e.g., *FRAME_TYPE_DATA_VEHICLE_SPEED*). The variable-length *PAYLOAD* carries the main information: for control frames it includes a 1-byte sub-command for Slave mode selection and for data frames, it contains the emulated sensor data, potentially including a sequence ID. To ensure data integrity, a software-calculated 8-bit Cyclic Redundancy Check (CRC-8) which is also covering the essential frame components, is appended before the concluding *END_BYTE* (e.g., 0x55).

START_BYTE (1 Byte) (0xAA)	**LENGTH** (1 Byte) (Length = Payload + 1)	**TYPE** (1 Byte)	**PAYLOAD** (0 to N Bytes, e.g., up to MAX_FRAM_SIZE – 5 Overhead bytes)	**CRC - 8** (1 Byte)	**END_BYTE** (1 Byte) (0x55)

Figure 3-6: Custom SPI Frame Structure

The sensor data is internally managed using specific C *struct* definition, that is designed with packing attributes (e.g., *__attribute__((packed))*) to ensure direct memory compatibility with the SPI frame payload. Structures like *VehicleSpeedData* (containing a float), *ImuGyroData* (containing three int16_t values), and *ImuAccelData* (containing three float values) are included that represent the different types of sensor information the Slave can emulate. During the formation of the SPI payload, these structures combined with a sequence identifier facilitate organized data handling within the firmware.

3.2.3.2 High-Level System Communication Sequence

The end-to-end communication sequence is the interaction from the external host through the Master and Slave boards. A typical interaction cycle begins when the dSPACE HIL simulator or a Python test script sends a UDP packet to the Master STM32 board. The LwIP stack on the Master processes this packet, and its UDP receive callback extracts the command or data, storing the sender's network details. Subsequently, the Master firmware constructs the custom SPI command frame, including any necessary payload (like a sub-command or data for the slave), a sequence ID, and a CRC-8. The Master starts a two-way SPI DMA transaction by activating the NSS line using GPIO PG4 and sends this frame to the Slave.

Slave when it detects the NSS assertion, receives the SPI frame through its circular DMA mechanism. DMA interrupt callbacks trigger the parsing of the received frame that includes the validation of CRC and the interpretation of the command. Based on the command received from the Master board, the Slave updates its internal state (e.g., the type of sensor data to stream) and prepares an appropriate SPI response. This response, whether an ACK/NACK frame or a data frame containing emulated sensor values (e.g., speed, gyroscope, accelerometer data), an echoed sequence ID, and a new CRC-8—is loaded into the Slave's circular SPI TX DMA buffer. This prepared data is then clocked out by the Master during the same or a subsequent SPI transaction initiated by the Master (common in polling or continuous data streaming scenarios).

Upon completion of its SPI DMA transaction, the Master's SPI completion callback is called. It de-asserts NSS, parses the frame received from the Slave, verifies its integrity (CRC and sequence ID), and extracts the payload. If valid sensor data or an expected acknowledgment is received, this information is then packaged into a UDP datagram by the Master and transmitted back to the originating host; hence, in this way, the communication loop is completed. For continuous data acquisition, the Master may periodically send poll frames or resend control commands to the Slave, with internal timeout and retry mechanisms to handle potential SPI communication disruptions.

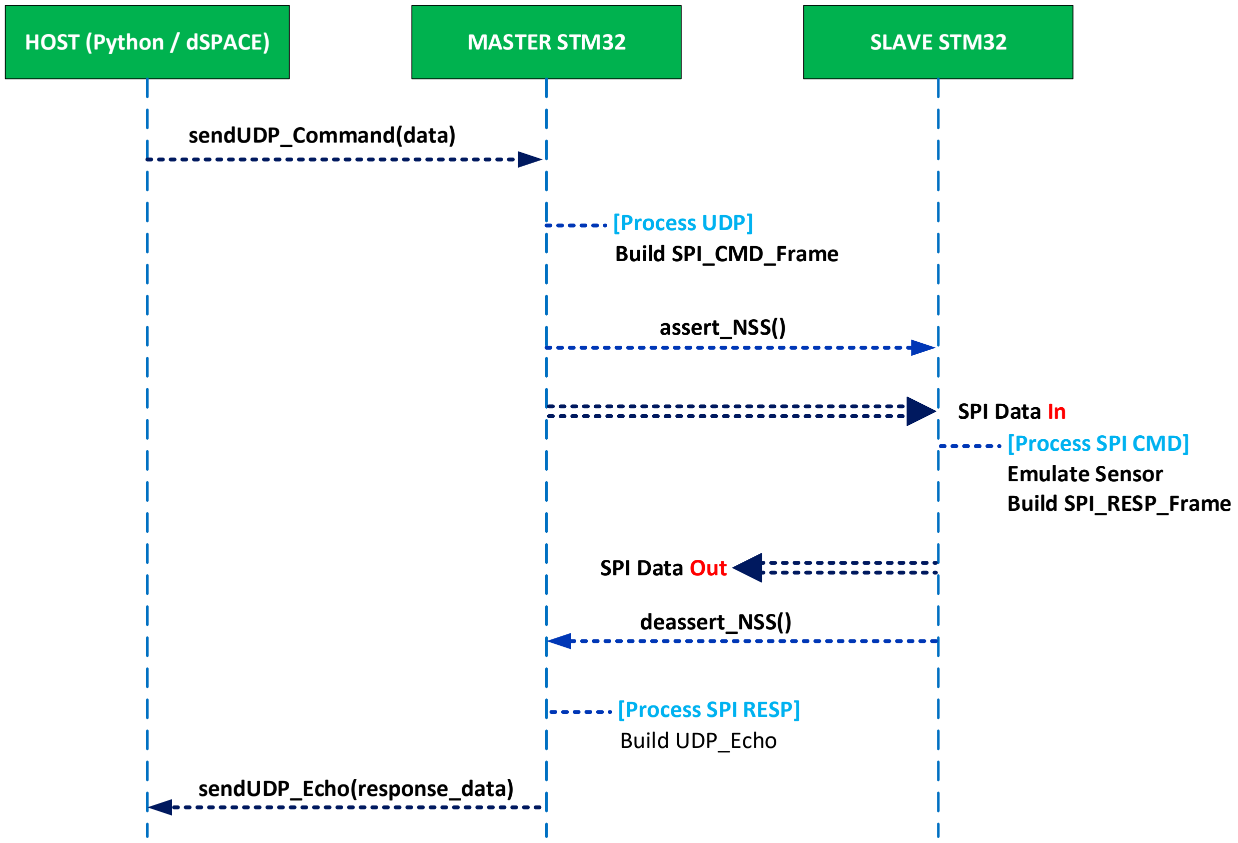


Figure 3-7: High-Level Communication Sequence Diagram for a Command-Response Cycle

3.3 Simulink and dSPACE Package

The dSPACE HIL simulation, centered around the SCALEXIO platform and DS6001 processor board, was used to generate real-time UDP commands for the STM32 Master board and observe the behavior of the system. This simulation was achieved through a specifically designed Simulink model and then compiled into a dSPACE real-time application and was managed via a custom dSPACE ControlDesk interface.

3.3.1 dSPACE Real-Time Application Configuration and Deployment

A new project was created in dSPACE ConfigurationDesk (e.g., named "Speed_Echo_CD," "Sensor_Mimic," or "Speed_Echo_Latency"). The SCALEXIO hardware platform was then defined within this project by adding a "SCALEXIO Rack" and subsequently a "LabBox (8-slot)" to mirror the physical setup to be used.

Inside the "Signal Chain" view of dSPACE ConfigurationDesk, the necessary function blocks for Ethernet-based UDP communication were added, as shown in Figure 3-8: dSPACE ConfigurationDesk Function Blocks for UDP-Based Interaction. This included an "Ethernet Setup" block, a "UDP Transmit" block, and a "UDP Receive" block. The "Ethernet Setup" block was configured with the local IP address 169.254.32.8 and subnet mask 255.255.0.0 for the DS6001 board and was mapped to its physical Eth0_1 Ethernet port. The "UDP Transmit" block was configured to send data to the STM32 Master board at IP address 169.254.32.100 on UDP port 50000, with a maximum vector size (e.g., 4 bytes for simple speed data or 1 byte for sub-commands). The "UDP Receive" block was configured to listen on local port 50000 for data from the STM32 Master, with an appropriate vector size (e.g., 4 bytes for echoed speed or up to 13 bytes for structured sensor data). The "Model port data type" under "Propagation" was set to "Inherited" for both UDP Transmit and Receive blocks in order to ensure that the data exchanged with the subsequent Simulink model interface would be as a uint8[] byte array.

Once these hardware function blocks were configured in the Signal Chain, all these blocks were selected, and a Simulink model was created using the "Generate New Simulink Model Interface" option. This automatically created a new Simulink model file (which, for e.g., was named "Speed_Echo.slx"). This generated model contained the basic blocks (e.g., "UDP Receive (1)," "UDP Transmit (1)_Out," and "UDP Transmit (1)_In") as depicted in Figure 3-9: Simulink Model automatically created via dSPACE ConfigurationDesk, which serve as the bridge between the logical model to be implemented and the physical I/O defined in dSPACE ConfigurationDesk. This auto-generated file is then used as a starting point for implementing specific tests in Simulink.

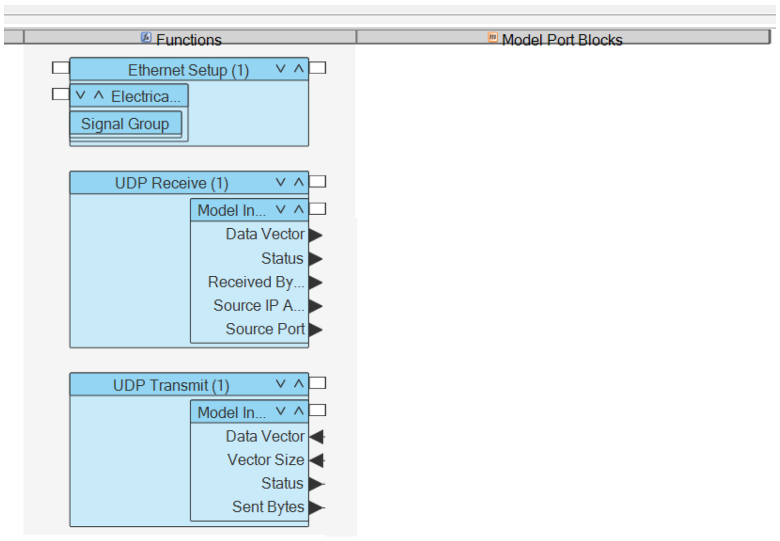


Figure 3-8: dSPACE ConfigurationDesk Function Blocks for UDP-Based Interaction

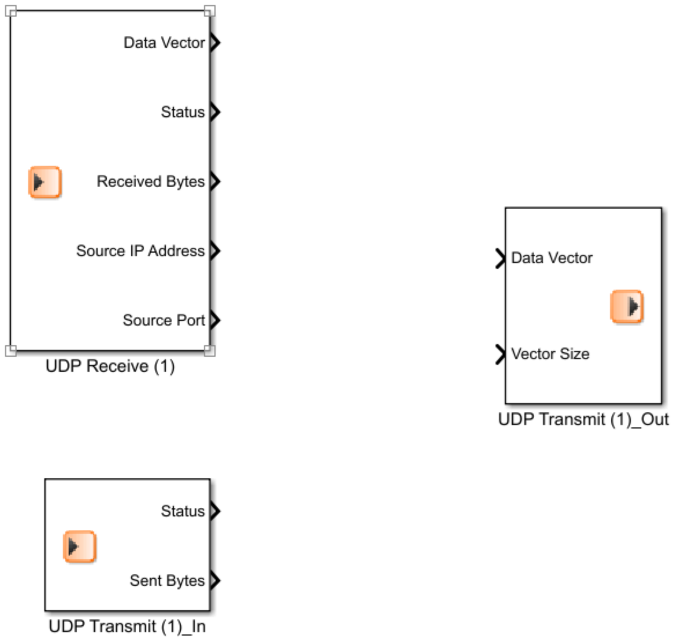


Figure 3-9: Simulink Model automatically created via dSPACE ConfigurationDesk

3.3.2 Simulink Models for UDP-Based Interaction

The HIL simulator’s functionality is implemented using a MATLAB/Simulink model. This model is designed to emulate the external system that sends input commands to the STM32 Master board and also receive and process data returned from the STM32 system via UDP. Once this base Simulink model interface was created by dSPACE ConfigurationDesk, several Simulink models were developed to test the objectives for the Ethernet-SPI Bridge. These models were saved with specific names (e.g., "Speed_Echo_Latency.slx", "Speed_Echo_CD.slx" and "Sensor_Mimic.slx") and subsequently integrated back into the dSPACE ConfigurationDesk project.

3.3.2.1 Model 1: (Speed_Echo_Latency) – Internal Speed Generation and Echo Latency Verification

The model was not only designed for the end-to-end verification of UDP paths but also to perform a real-time measurement of the round-trip time (RTT). Its purpose was to generate dynamic speed data, transmit it to the STM32-based Ethernet-SPI Bridge, receive the corresponding echo, and precisely calculate the communication latency within the HIL environment. The model architecture comprises three main functional paths which are data transmission path, data reception path, and latency calculation path.

For data transmission, a MATLAB function block, *GenerateSpeed*, was implemented to produce random floating-point speed values (e.g., within a range from 60.0 to 80.0 kph). The output of *GenerateSpeed* was then connected to another MATLAB function, *PackFloatToBytes*, which converted the single-precision float into a 4-element uint8 vector. This byte vector was connected to the "Data Vector" Inport of the "UDP Transmit (1)_Out" interface block. A constant block providing the vector size (e.g., 4 for the float data, which is 4 bytes) was connected to the "Vector Size" input of this UDP Transmit (1)_Out interface block.

For data reception, the "Data Vector" outport from the "UDP Receive (1)" that outputs the received uint8 array is connected to a MATLAB function block named "*UnpackBytesToFloat.*" This function converted the received 4-byte array back into a single-precision float, and its output, which represented the speed echoed, was connected to a *Data Store Write* block (*Speed_Echo*) for storing values and real-time visualization.

For calculating the communication latency, a MATLAB function block named "*ComputeLatencyWithBuffer*" was utilized. This function was designed for RTT measurement and takes three inputs: the original "*sentSpeed*" from the "*GenerateSpeed*" block, the received "*echoSpeed*" from the "*UnpackBytesToFloat*" block, and "*t_now*," a timestamp provided by a Simulink Clock block. The function maintains a continuous circular buffer of recent sent values along with their timestamps. Upon receiving an *echoSpeed*, it searches this buffer for a matching *sentSpeed* and calculates the latency. The resulting latency output, in milliseconds, was connected to a Data Store Write block named *EchoLatency_ms* to make it accessible for real-time monitoring in dSPACE ControlDesk. This model is depicted in Figure 3-10: Simulink Model for Randomly Generated Speed Values and Latency Calculation.

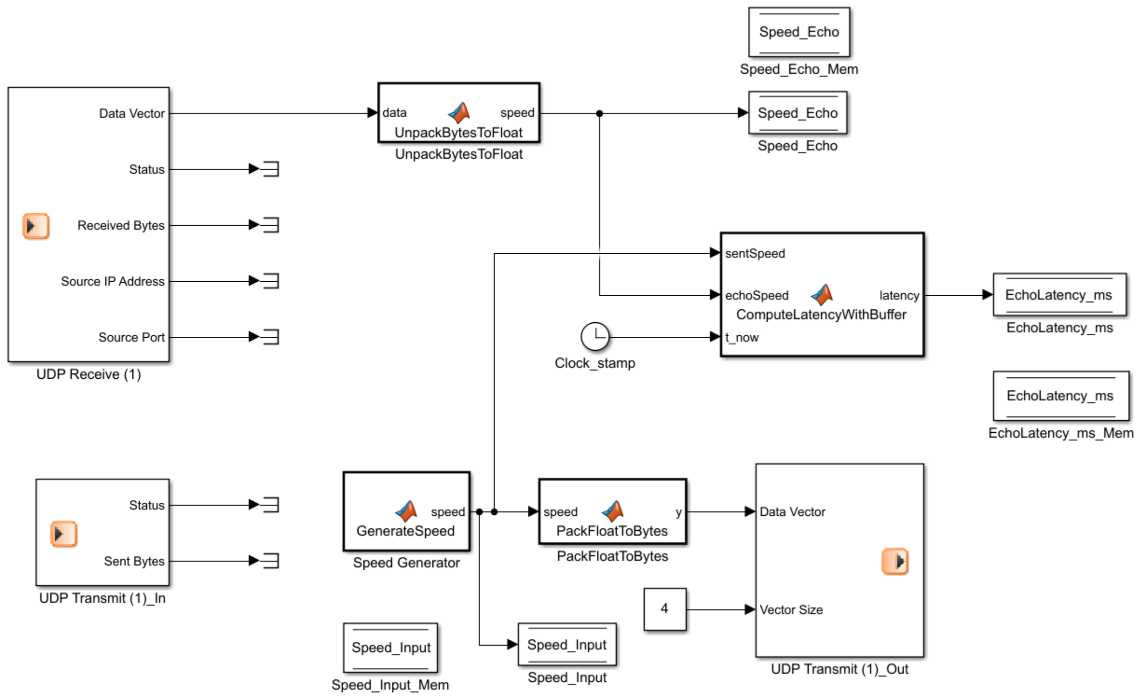


Figure 3-10: Simulink Model for Randomly Generated Speed Values and Latency Calculation

3.3.2.2 Model 2: (Speed_Echo_CD) – dSPACE ControlDesk Driven Speed Control and Monitoring

The model shown in the Figure 3-11: Simulink Model for Controlling Speed and Monitoring in ControlDesk shows how we can control *Speed_Input* using a slider/knob in dSPACE ControlDesk. A constant of type "Single" and value "1" can be accessed in dSPACE ControlDesk to attach a knob/slider to change the *Speed_Input* values on our own. The output function "*UnpackBytesToFloat*" is connected to the "*Data Store Write*" block that is linked with the "*Data Store Memory*" block, which can store the values in a variable called "*Speed_Echo*." This variable can be accessed in dSPACE ControlDesk to attach a graph, display, etc., to it for monitoring the values received.

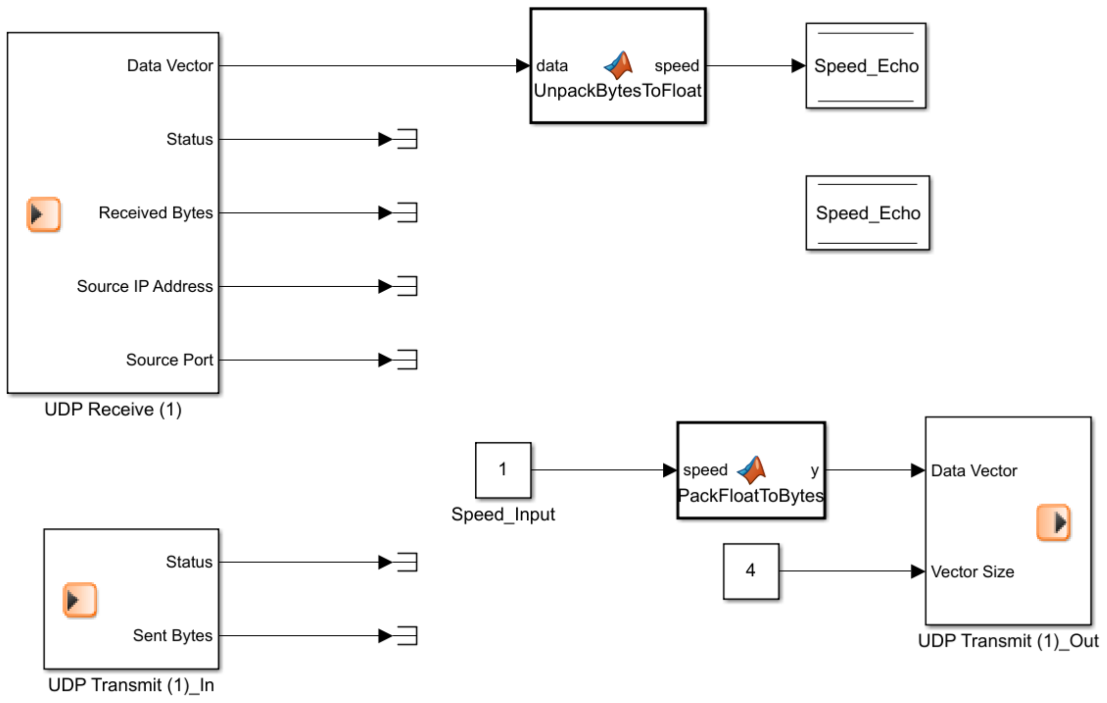


Figure 3-11: Simulink Model for Controlling Speed and Monitoring in ControlDesk

3.3.2.3 Model 3: (Sensor_Mimic) – dSPACE ControlDesk Driven Speed Control and Monitoring

This Simulink model was designed to allow a user to send different commands to the STM32 Master via dSPACE ControlDesk GUI. These are 1-byte sub-commands that are used to direct the STM32 Slave, which emulates a multi-functional sensor, to switch between different data streaming modes (e.g., Idle, Vehicle Speed, IMU Gyroscope, and IMU Accelerometer). The Simulink model then receives the UDP data packets as a response from STM32 Slave via STM32 Master, parses this information, and formats it into a human-readable string for real-time display in dSPACE ControlDesk.

A Simulink constant block named *SubCmd Control* is configured to send a uint8 value and can be connected to a ON/OFF-button or drop-down menu bar to trigger a specific sub-command such as *SUB_CMD_SET_STREAM_IDLE (0x00)*, *SUB_CMD_SET_STREAM_SPEED (0x01)*, *SUB_CMD_SET_STREAM_GYRO (0x02)*, or *SUB_CMD_SET_STREAM_ACCEL (0x03))*. These commands from this constant block are connected to the "Data Vector" input of the "UDP Transmit (1)_Out" interface block. Another constant block with data type Boolean and value “1” is connected to the "Vector Size" input of the "UDP Transmit (1)_Out" block, ensuring only the single command byte is transmitted as the UDP payload.

The "UDP Receive (1)" interface block in Simulink is configured with a "Vector size" sufficient enough to accommodate the largest possible UDP payload received from the STM32 Master (e.g., 13 bytes, comprising a 1-byte Slave SPI Frame Type + 12 bytes for IMU Accelerometer float data). The output of this UDP receive block (Data Vector) is connected to a MATLAB function block named *UnpackFrame*, which interprets the raw byte data (uint8) received from the STM 32 Master board. That data contains frame type identifier (First Byte) and a payload (e.g. float Speed, int16 Gyro, float Accel). This function checks for the type of data the frame holds and based on that they are then converted to int32 (to make Simulink codegen-friendly), and passed out as out1, out2, and out3 to another MATLAB function block named *FormatResponse*. This *FormatResponse* function builds a displayable message by manually assembling text using fixed-size uint8 arrays. The logic performed by these functions are as follows:

- Extracted the first byte of the received UDP payload as the *Slave_SPI_Frame_Type* (e.g., 0xA1 for Speed, 0xA2 for Gyro, 0xB0 for ACK, etc.).
- Based on this *Slave_SPI_Frame_Type*, it parsed the subsequent bytes of the UDP payload into their respective data types:
  - For *FRAME_TYPE_DATA_VEHICLE_SPEED (0xA1)*: The next 4 bytes were typecast to a single-precision float (speed in kph).
  - For *FRAME_TYPE_DATA_IMU_GYRO (0xA2)*: The next 6 bytes were typecast to three int16_t values (raw gyro x, y, z).
  - For *FRAME_TYPE_DATA_IMU_ACCEL (0xA3)*: The next 12 bytes were typecast to three single-precision floats (accel x, y, z in g).
  - For *FRAME_TYPE_SLAVE_ACK (0xB0)* or *FRAME_TYPE_SLAVE_NACK (0xB1)*: The next 2 bytes were interpreted as the echoed sub-command and a status byte.
- This parsed data is then formatted into a string which can be read by human (e.g., "Speed: XX.YY kph", "Gyro: X=..., Y=..., Z=...", "ACK: SubCmd=0xZZ Status=OK/FAIL", or "Unknown Frame: Type=0xAA Payload=...") as a fixed-length uint8 array. This formatting was done using array manipulations and ASCII conversions for codegen compatibility, avoiding non-embeddable string functions.

This uint8 array string output was connected to a Gain block (unity gain) and the output as the *Slave_Response* for display. The output of the function "*FrameResponse*" can be accessed in dSPACE ControlDesk variable array, display, etc., for monitoring the data received.

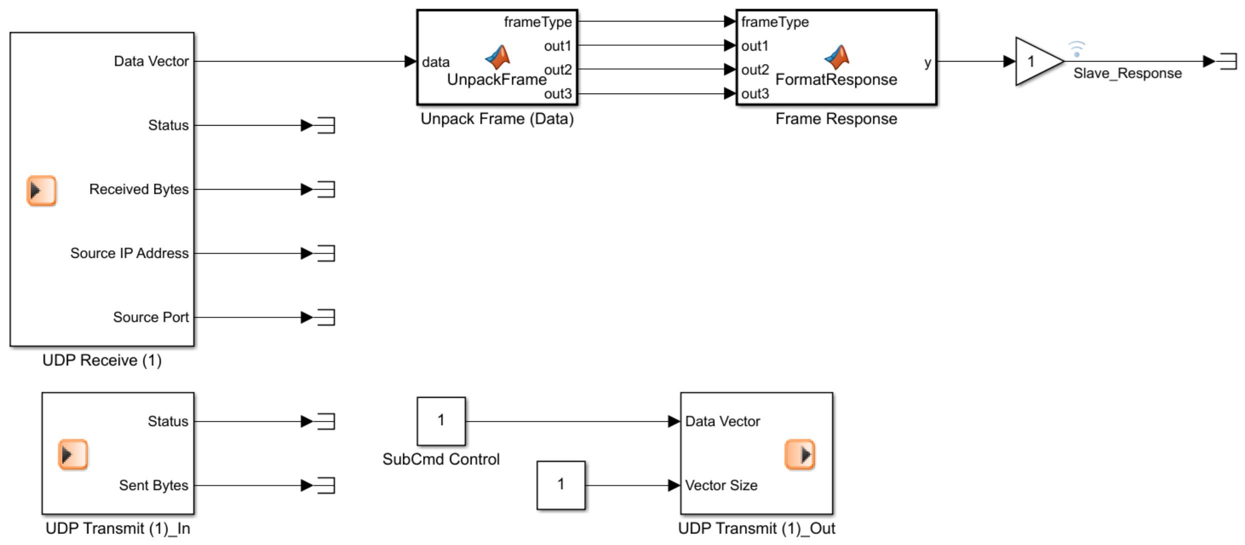
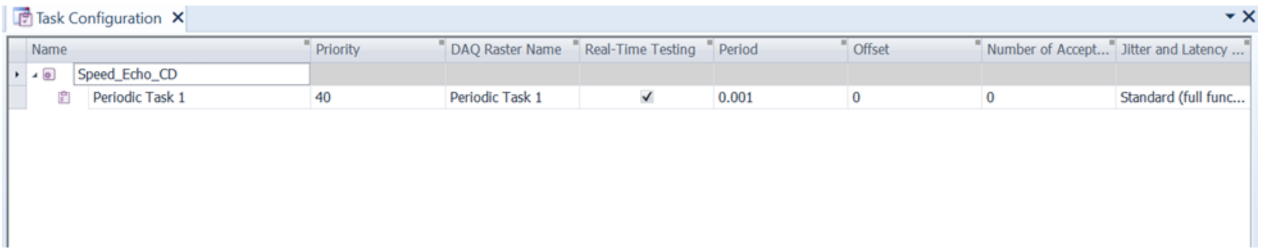


Figure 3-12: Simulink Model for Slave as a multi-functional sensor

3.3.3 Real-Time Application Build and Deployment

After the development and saving of the respective Simulink, the workflow returned to dSPACE ConfigurationDesk to integrate these models and build the final real-time application. In “Models” tab of Signal Chain, the Simulink model was selected and was updated within dSPACE ConfigurationDesk by clicking “Analyze”. By doing so, the logical implementation done in the Simulink file were mapped within dSPACE ConfigurationDesk as well. This mapping step is crucial as it establishes the direct data flow between the Simulink model's computational logic and the physical Ethernet I/O of the SCALEXIO system. The execution rate of the real-time application, set as 1-ms, is configured in the Task Configuration settings within dSPACE ConfigurationDesk.



Name	Priority	DAQ Raster Name	Real-Time Testing	Period	Offset	Number of Accept...	Jitter and Latency ...
Speed_Echo_CD							
Periodic Task 1	40	Periodic Task 1	✓	0.001	0	0	Standard (full func...

Figure 3-13: dSPACE ConfigurationDesk - Task Configuration

With the model-function mapping and task configuration done, the real-time application was generated using the "Start Build (CTRL+ALT+B)" command from dSPACE ConfigurationDesk. This process compiled the Simulink model and integrated it with the dSPACE ConfigurationDesk hardware and tasking settings, producing the necessary executable files (.rta), interface definitions (.sdf), and dSPACE ControlDesk layout files (.a21) for deployment on the SCALEXIO DS6001 processor board. The successful build was confirmed by reviewing the build log and the generated artifacts in the project browser as shown in Figure 3-14: dSPACE ConfigurationDesk Build Results.

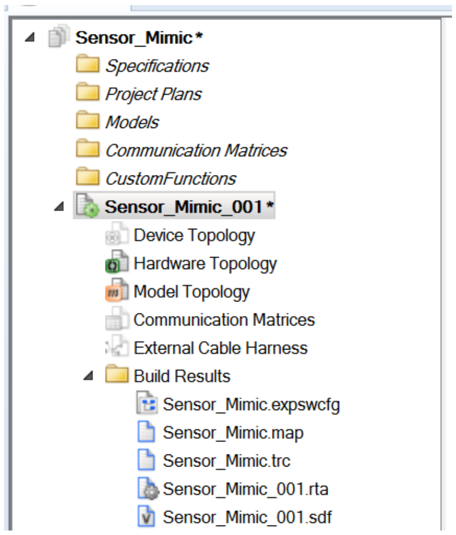
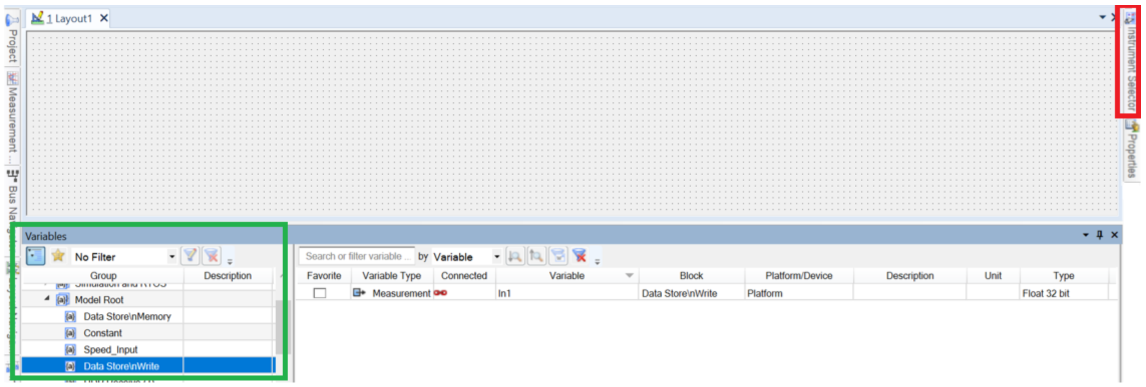


Figure 3-14: dSPACE ConfigurationDesk Build Results

3.3.4 Experiment Control with dSPACE ControlDesk

The final stage of the HIL application setup involves utilizing dSPACE ControlDesk to load and execute the compiled real-time application—the .sdf file that was created using dSPACE ConfigurationDesk and Simulink—on the SCALEXIO platform. dSPACE ControlDesk serves as the primary Human-Machine Interface (HMI) that provides a user-friendly environment that allows for interactive experiment control and real-time data visualization of the application, which is crucial for comprehensive HIL testing and validation.

The process of creating the dSPACE ControlDesk interface begins after the Simulink model is built and its variables are available within the dSPACE ControlDesk environment. As shown in Figure 3-15: dSPACE ControlDesk Layout - Variable and Instrument Selector Pane, the "Variables" pane (highlighted in the green box) lists the accessible signals and parameters from the loaded real-time application (e.g., "Sensor_Mimic.sdf", "Speed_Echo_Latency.sdf" or "Speed_Echo_CD.sdf"). This includes a parameter named *Speed_Input* (configured as a constant in Simulink) and a measurement *Speed_Echo* (configured as a Data Store Write in Simulink for displaying echo output). The "Instrument Selector" tab (highlighted in the red box) provides the list of the instruments that can be dragged onto the layout and connected to these variables.



Group	Description
Model Root	
Data Store\nMemory	
Constant	
Speed_Input	
Data Store\nWrite	

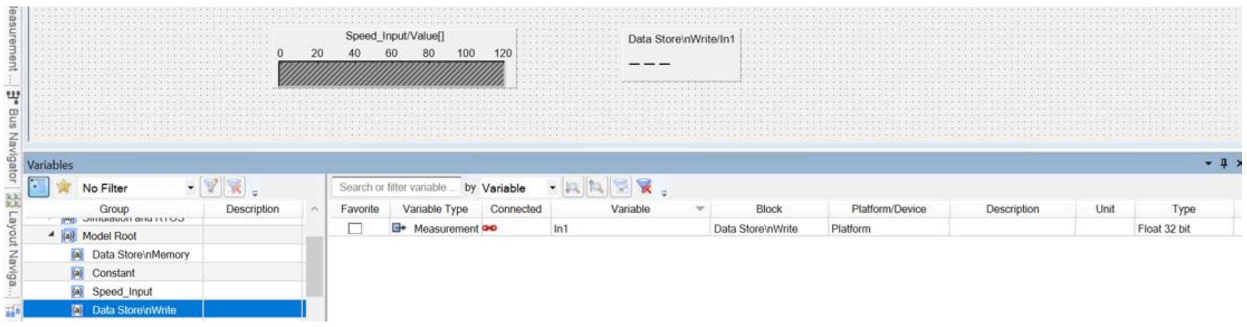
Favorite	Variable Type	Connected	Variable	Block	Platform/Device	Description	Unit	Type
☐	Measurement		In1	Data Store\nWrite	Platform			Float 32 bit

Figure 3-15: dSPACE ControlDesk Layout - Variable and Instrument Selector Pane

For the *Speed_Echo* test, the dSPACE ControlDesk layout was designed for controlling *Speed_Input* and *Speed_Echo* (Data Store Write) variables. A slider and a numeric display were dragged and dropped from the Instrument Selector into the layout, with the *Speed_Input* variable assigned to the slider, and the "Data Store Write" variable assigned to the number display, as shown in Figure 3-16: dSPACE ControlDesk Layout - Variable and Instrument Integrated. This layout allowed the operator to vary the speed value (via slider) that is being sent via UDP to the STM32 system and visualize the echoed speed value in the numerical display.

For the Speed_Echo_latency test, additional instruments were added in the dSPACE ControlDesk layout to perform the analysis. In addition to visualizing the sent and echoed speed signals, dedicated instruments were added for latency monitoring. A numeric display was connected to the *EchoLatency_ms* "Data Store Write" variable to show the instantaneous RTT value, while a time-plot graph was also linked to this variable to demonstrate the latency over time. This layout enabled the observation of latency, jitter, and the system's overall response.

Similarly, for the Sensor_Mimic test, the dSPSACE ControlDesk layout was designed to use the instruments like ON/OFF-button, radio buttons or pushbuttons. This instrument is to be linked to the variable *SubCmd Control*, which allows the user to send the 1-byte sub-commands to command the STM32 Slave to stream different types of sensor data. The resulting responses from the Slave, processed and formatted by the Simulink model into a displayable string (e.g., *FrameResponse* function output), would be either displayed in the dSPACE ControlDesk or stored in the form of a .csv file.



Group	Description
Model Root	
Data Store\nMemory	
Constant	
Speed_Input	
Data Store\nWrite	

Favorite	Variable Type	Connected	Variable	Block	Platform/Device	Description	Unit	Type
☐	Measurement		In1	Data Store\nWrite	Platform			Float 32 bit

Figure 3-16: dSPACE ControlDesk Layout - Variable and Instrument Integrated

Chapter 4

Implementation and Methodology

4.1 Ethernet Phase: STM32 UDP Communication

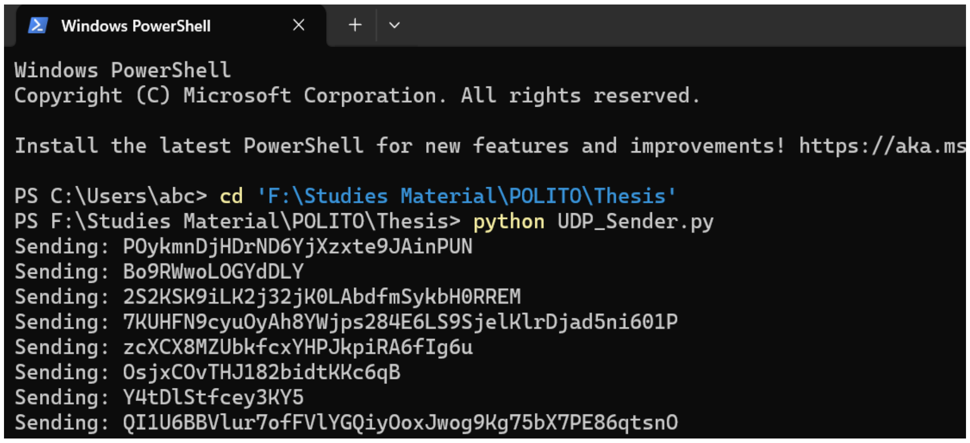
The initial development phase was to establish a robust UDP communication capability using a single STM32F207ZG board, which later on served as the Master in the complete system. This development phase involved configuring the LwIP stack for bare-metal operation, which led to the development of the firmware to handle both UDP server and client functionalities. These tests were carried out using Python scripts running over the Windows PowerShell in the PC and a network utility application, Packet Sender. The stepwise implementation of the Ethernet development phase is as follows.

4.1.1 STM32 as a UDP Server: Command Reception and Processing

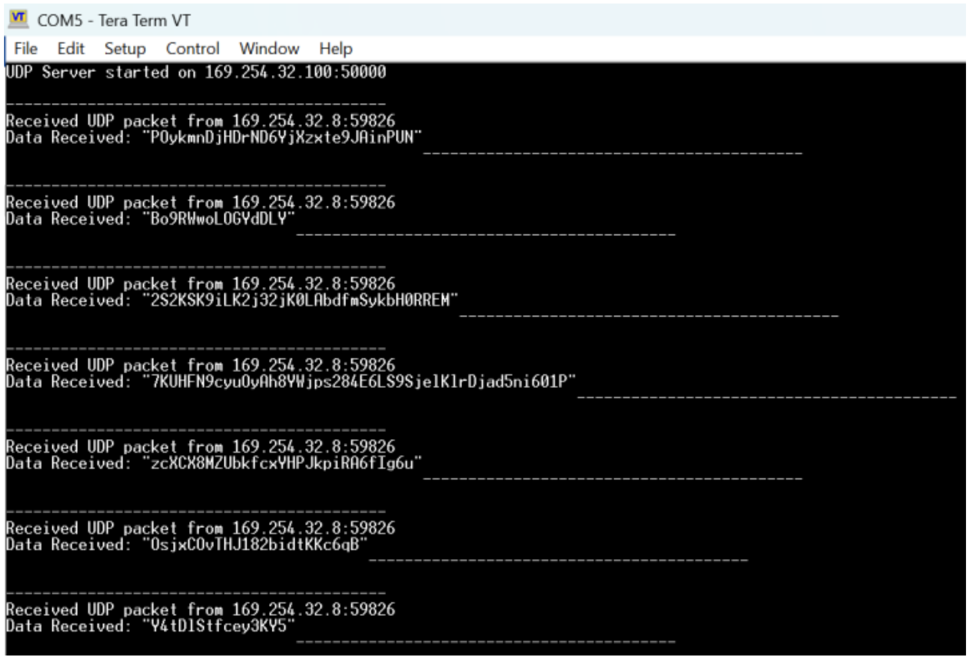
Initially, the STM32F207ZG board was configured to operate as a UDP server so that its UDP reception capabilities could be validated. This setup involved the initialization of the LwIP stack inside CubeMX for the bare-metal environment, and firmware was implemented to listen to the incoming UDP packets at a predefined static IP address (e.g., 169.254.32.100) and UDP port (e.g., 50000). The core of this server was inside the *udp_server_init* function, which established the UDP Protocol Control Block (PCB), and a dedicated *udp_receive_callback* function, which was triggered by the LwIP stack upon the arrival of a UDP packet on the designated port.

4.1.1.1 Basic Data Reception and Display

The aim of the first test was to confirm basic data reception and parsing. A Python script was developed to send variable-length UDP packets having data type string to the STM32 board. Inside the *udp_receive_callback* function of STM32 code, these incoming payloads were extracted from the *pbuf* structure. To ensure safe handling and prevent buffer overflows, the received data was copied into an array of local characters that null-terminated the string. This processed string, often appended with a packet counter for sequence verification, was then displayed on a serial terminal (Tera Term, connected via UART) for immediate visual confirmation of successful reception and data integrity. Besides that, Wireshark was also used on the PC to monitor the UDP traffic, verifying packet transmission and reception at the network level.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms

PS C:\Users\abc> cd 'F:\Studies Material\POLITO\Thesis'
PS F:\Studies Material\POLITO\Thesis> python UDP_Sender.py
Sending: POykmnDjHDrND6YjXzxte9JAinPUN
Sending: Bo9RWwoLOGYdDLY
Sending: 2S2KSK9iLK2j32jK0LAbdfmSykbH0RREM
Sending: 7KUHFN9cyuOyAh8YWjps284E6LS9SjelKlrDjad5ni601P
Sending: zcXCX8MZUbkfcxYHPJkpiRA6fIg6u
Sending: OsjxCOvTHJ182bidtKKc6qB
Sending: Y4tDlStfcey3KY5
Sending: QI1U6BBVlur7ofFVlYGQiyOoxJwog9Kg75bX7PE86qtsnO
```

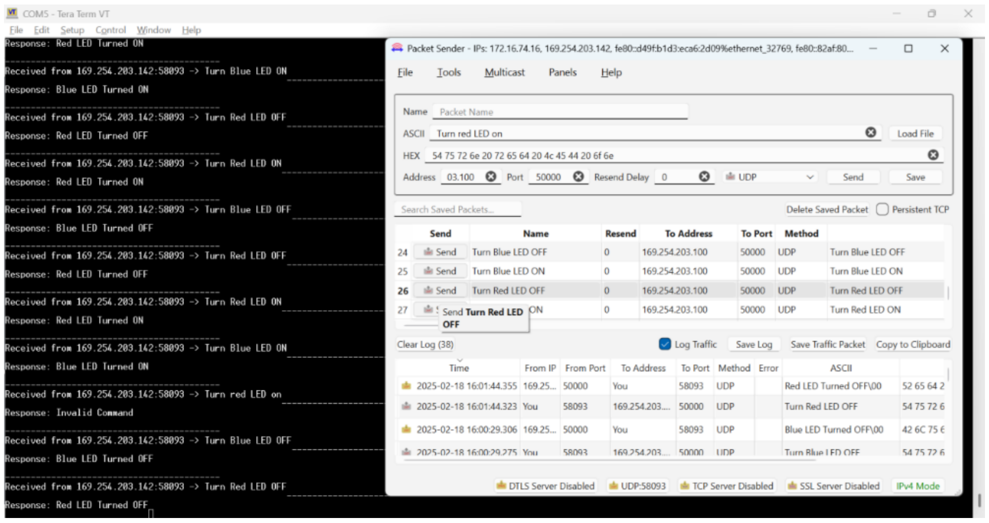


```
UDP Server started on 169.254.32.100:50000

----------------------------------------
Received UDP packet from 169.254.32.8:59826
Data Received: "POykmnDjHDrND6YjXzxte9JAinPUN"
                                              ----------------------------------------

----------------------------------------
Received UDP packet from 169.254.32.8:59826
Data Received: "Bo9RWwoLOGYdDLY"
                                ----------------------------------------

----------------------------------------
Received UDP packet from 169.254.32.8:59826
Data Received: "2S2KSK9iLK2j32jK0LAbdfmSykbH0RREM"
                                                  ----------------------------------------

----------------------------------------
Received UDP packet from 169.254.32.8:59826
Data Received: "7KUHFN9cyuOyAh8YWjps284E6LS9SjelKlrDjad5ni601P"
                                                               ----------------------------------------

----------------------------------------
Received UDP packet from 169.254.32.8:59826
Data Received: "zcXCX8MZUbkfcxYHPJkpiRA6fIg6u"
                                              ----------------------------------------

----------------------------------------
Received UDP packet from 169.254.32.8:59826
Data Received: "OsjxCOvTHJ182bidtKKc6qB"
                                        ----------------------------------------

----------------------------------------
Received UDP packet from 169.254.32.8:59826
Data Received: "Y4tDlStfcey3KY5"
                                ----------------------------------------
```

Figure 4-1: Basic UDP Transmission (Top) and Reception (Bottom)

4.1.1.2 Command Processing and Acknowledgment (LED Control)

The firmware was enhanced to simulate command-response interaction, and this functionality was achieved by sending specific string commands for controlling on-board hardware. These commands were sent using the Python script or a network utility like "Packet Sender." Commands such as "Turn RED LED ON" or "Turn BLUE LED OFF" were transmitted to the STM32 server. String comparison logic (e.g., using strcmp) was added to the *udp_receive_callback* function so that these commands could be parsed. After receiving a valid command, the firmware would set/reset the appropriate GPIO pins to control the state of these STM32 board's LEDs (e.g., LD1 - Green, LD2 - Blue, LD3 - Red). Subsequently, a response string such as "RED LED TURNED ON" or "Invalid Command" was transmitted back to the sender via UDP as an acknowledgment of the action taken or an error status. This test validated not only command parsing but also the STM32's ability to act on commands and send UDP responses.

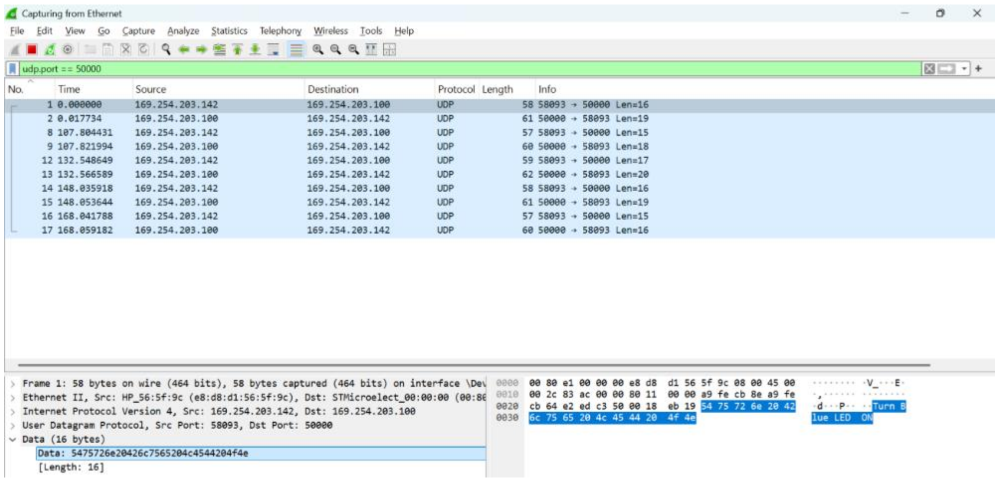


```
Response: Red LED Turned ON
-----------------------------------------
Received from 169.254.203.142:58093 -> Turn Blue LED ON
Response: Blue LED Turned ON
-----------------------------------------
Received from 169.254.203.142:58093 -> Turn Red LED OFF
Response: Red LED Turned OFF
-----------------------------------------
Received from 169.254.203.142:58093 -> Turn Red LED ON
Response: Red LED Turned ON
-----------------------------------------
Received from 169.254.203.142:58093 -> Turn Blue LED OFF
Response: Blue LED Turned OFF
-----------------------------------------
Received from 169.254.203.142:58093 -> Turn Red LED OFF
Response: Red LED Turned OFF
-----------------------------------------
Received from 169.254.203.142:58093 -> Turn Red LED ON
Response: Red LED Turned ON
-----------------------------------------
Received from 169.254.203.142:58093 -> Turn Blue LED ON
Response: Blue LED Turned ON
-----------------------------------------
Received from 169.254.203.142:58093 -> Turn red LED on
Response: Invalid Command
-----------------------------------------
Received from 169.254.203.142:58093 -> Turn Blue LED OFF
Response: Blue LED Turned OFF
-----------------------------------------
Received from 169.254.203.142:58093 -> Turn Red LED OFF
Response: Red LED Turned OFF
```

Name: Packet Name
ASCII: Turn red LED on
HEX: 54 75 72 6e 20 72 65 64 20 4c 45 44 20 6f 6e
Address: 03.100 Port: 50000 Resend Delay: 0 UDP

	Send	Name	Resend	To Address	To Port	Method	
24	Send	Turn Blue LED OFF	0	169.254.203.100	50000	UDP	Turn Blue LED OFF
25	Send	Turn Blue LED ON	0	169.254.203.100	50000	UDP	Turn Blue LED ON
26	Send	Turn Red LED OFF	0	169.254.203.100	50000	UDP	Turn Red LED OFF
27	Send Turn Red LED OFF	ON	0	169.254.203.100	50000	UDP	Turn Red LED ON

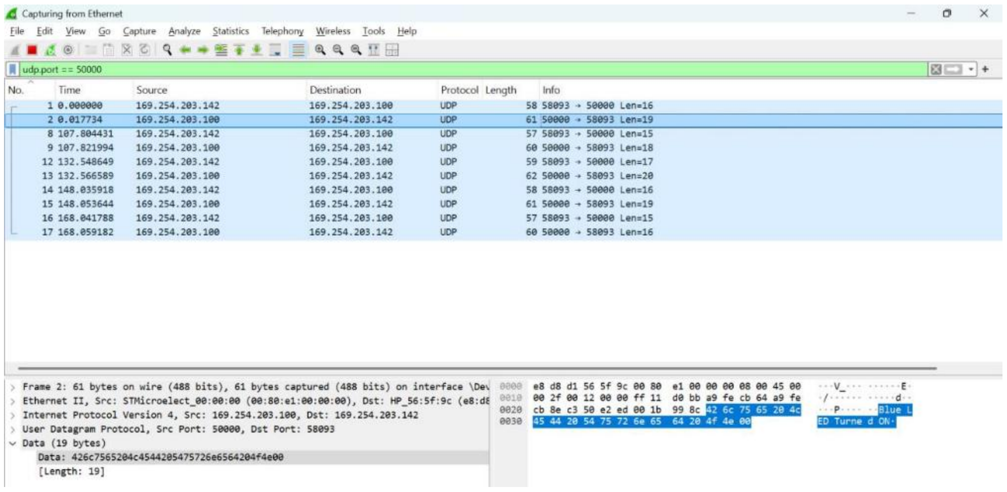
Time	From IP	From Port	To Address	To Port	Method	Error	ASCII	
2025-02-18 16:01:44.355	169.25...	50000	You	58093	UDP		Red LED Turned OFF\00	52 65 64 2
2025-02-18 16:01:44.323	You	58093	169.254.203....	50000	UDP		Turn Red LED OFF	54 75 72 6
2025-02-18 16:00:29.306	169.25...	50000	You	58093	UDP		Blue LED Turned OFF\00	42 6C 75 6
2025-02-18 16:00:29.275	You	58093	169.254.203....	50000	UDP		Turn Blue LED OFF	54 75 72 6


Capturing from Ethernet
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

udp.port == 50000

```
No.  Time        Source           Destination      Protocol Length Info
   1 0.000000    169.254.203.142  169.254.203.100  UDP      58 58093 → 50000 Len=16
   2 0.017734    169.254.203.100  169.254.203.142  UDP      61 50000 → 58093 Len=19
   8 107.804431  169.254.203.142  169.254.203.100  UDP      57 58093 → 50000 Len=15
   9 107.821994  169.254.203.100  169.254.203.142  UDP      60 50000 → 58093 Len=18
  12 132.548649  169.254.203.142  169.254.203.100  UDP      59 58093 → 50000 Len=17
  13 132.566589  169.254.203.100  169.254.203.142  UDP      62 50000 → 58093 Len=20
  14 148.035918  169.254.203.142  169.254.203.100  UDP      58 58093 → 50000 Len=16
  15 148.053644  169.254.203.100  169.254.203.142  UDP      61 50000 → 58093 Len=19
  16 168.041788  169.254.203.142  169.254.203.100  UDP      57 58093 → 50000 Len=15
  17 168.059182  169.254.203.100  169.254.203.142  UDP      60 50000 → 58093 Len=16
```

```
> Frame 1: 58 bytes on wire (464 bits), 58 bytes captured (464 bits) on interface \Dev
> Ethernet II, Src: HP_56:5f:9c (e8:d8:d1:56:5f:9c), Dst: STMicroelect_00:00:00 (00:80
> Internet Protocol Version 4, Src: 169.254.203.142, Dst: 169.254.203.100
> User Datagram Protocol, Src Port: 58093, Dst Port: 50000
v Data (16 bytes)
    Data: 5475726e20426c7565204c4544204f4e
    [Length: 16]

0000  00 80 e1 00 00 00 e8 d8  d1 56 5f 9c 08 00 45 00   ········ ·V_···E·
0010  00 2c 83 ac 00 00 80 11  00 00 a9 fe cb 8e a9 fe   ·,······ ········
0020  cb 64 e2 ed c3 50 00 18  eb 19 54 75 72 6e 20 42   ·d···P·· ··Turn B
0030  6c 75 65 20 4c 45 44 20  4f 4e                     lue LED  ON
```

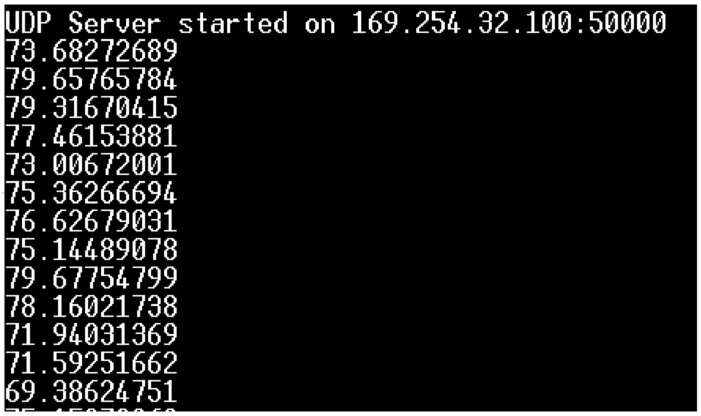

Capturing from Ethernet
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
udp.port == 50000

No.  Time        Source           Destination      Protocol Length Info
   1 0.000000    169.254.203.142  169.254.203.100  UDP      58 58093 → 50000 Len=16
   2 0.017734    169.254.203.100  169.254.203.142  UDP      61 50000 → 58093 Len=19
   8 107.804431  169.254.203.142  169.254.203.100  UDP      57 58093 → 50000 Len=15
   9 107.821994  169.254.203.100  169.254.203.142  UDP      60 50000 → 58093 Len=18
  12 132.548649  169.254.203.142  169.254.203.100  UDP      59 58093 → 50000 Len=17
  13 132.566589  169.254.203.100  169.254.203.142  UDP      62 50000 → 58093 Len=20
  14 148.035918  169.254.203.142  169.254.203.100  UDP      58 58093 → 50000 Len=16
  15 148.053644  169.254.203.100  169.254.203.142  UDP      61 50000 → 58093 Len=19
  16 168.041788  169.254.203.142  169.254.203.100  UDP      57 58093 → 50000 Len=15
  17 168.059182  169.254.203.100  169.254.203.142  UDP      60 50000 → 58093 Len=16

> Frame 2: 61 bytes on wire (488 bits), 61 bytes captured (488 bits) on interface \Dev
> Ethernet II, Src: STMicroelect_00:00:00 (00:80:e1:00:00:00), Dst: HP_56:5f:9c (e8:d8
> Internet Protocol Version 4, Src: 169.254.203.100, Dst: 169.254.203.142
> User Datagram Protocol, Src Port: 50000, Dst Port: 58093
v Data (19 bytes)
    Data: 426c7565204c4544205475726e6564204f4e00
    [Length: 19]

0000  e8 d8 d1 56 5f 9c 00 80  e1 00 00 00 08 00 45 00   ···V_··· ······E·
0010  00 2f 00 12 00 00 ff 11  d0 bb a9 fe cb 64 a9 fe   ·/······ ·····d··
0020  cb 8e c3 50 e2 ed 00 1b  99 8c 42 6c 75 65 20 4c   ···P···· ··Blue L
0030  45 44 20 54 75 72 6e 65  64 20 4f 4e 00            ED Turne d ON·
```

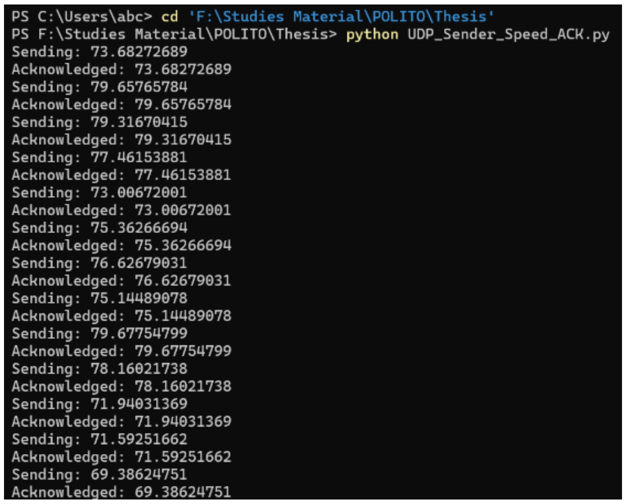
Figure 4-2: UDP Command and Response (Wireshark Capture and Tera Term Display)

4.1.1.3 Handling Numerical Data and Echoing (Car Speed Simulation)

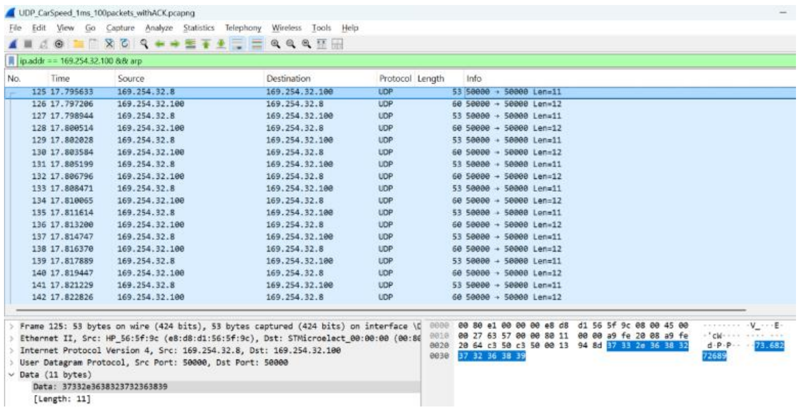
For the scenario that involved numerical data, a test was conducted where the STM32 server received UDP packets containing simulated car speed values. These speed values (floating-point numbers) were randomly generated by a Python script and were converted to string format before being transmitted. The *udp_receive_callback* in the STM32 code parsed the received string packets back into numerical values by using *strtod* (convert to a double). The received numerical speed was then displayed on the Tera Term serial terminal for verification, and the same value was echoed back to the Python script as an acknowledgment. This was tested with sequences of 100 and infinite packets at a 1-ms interval to assess the responsiveness and data handling capacity of the LwIP stack under high-frequency traffic. The successful reception and acknowledgment of these packets confirmed the system's ability to handle string-to-numerical data conversion and maintain integrity at a higher rate.



```
UDP Server started on 169.254.32.100:50000
73.68272689
79.65765784
79.31670415
77.46153881
73.00672001
75.36266694
76.62679031
75.14489078
79.67754799
78.16021738
71.94031369
71.59251662
69.38624751
```



```
PS C:\Users\abc> cd 'F:\Studies Material\POLITO\Thesis'
PS F:\Studies Material\POLITO\Thesis> python UDP_Sender_Speed_ACK.py
Sending: 73.68272689
Acknowledged: 73.68272689
Sending: 79.65765784
Acknowledged: 79.65765784
Sending: 79.31670415
Acknowledged: 79.31670415
Sending: 77.46153881
Acknowledged: 77.46153881
Sending: 73.00672001
Acknowledged: 73.00672001
Sending: 75.36266694
Acknowledged: 75.36266694
Sending: 76.62679031
Acknowledged: 76.62679031
Sending: 75.14489078
Acknowledged: 75.14489078
Sending: 79.67754799
Acknowledged: 79.67754799
Sending: 78.16021738
Acknowledged: 78.16021738
Sending: 71.94031369
Acknowledged: 71.94031369
Sending: 71.59251662
Acknowledged: 71.59251662
Sending: 69.38624751
Acknowledged: 69.38624751
```



UDP_CarSpeed_1ms_100packets_withACK.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
ip.addr == 169.254.32.100 && arp

No.  Time       Source          Destination     Protocol Length Info
125 17.795633  169.254.32.8    169.254.32.100  UDP      53 50000 → 50000 Len=11
126 17.797206  169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=12
127 17.798944  169.254.32.8    169.254.32.100  UDP      53 50000 → 50000 Len=11
128 17.800514  169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=12
129 17.802028  169.254.32.8    169.254.32.100  UDP      53 50000 → 50000 Len=11
130 17.803584  169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=12
131 17.805199  169.254.32.8    169.254.32.100  UDP      53 50000 → 50000 Len=11
132 17.806796  169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=12
133 17.808471  169.254.32.8    169.254.32.100  UDP      53 50000 → 50000 Len=11
134 17.810065  169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=12
135 17.811614  169.254.32.8    169.254.32.100  UDP      53 50000 → 50000 Len=11
136 17.813200  169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=12
137 17.814747  169.254.32.8    169.254.32.100  UDP      53 50000 → 50000 Len=11
138 17.816370  169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=12
139 17.817889  169.254.32.8    169.254.32.100  UDP      53 50000 → 50000 Len=11
140 17.819447  169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=12
141 17.821229  169.254.32.8    169.254.32.100  UDP      53 50000 → 50000 Len=11
142 17.822826  169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=12

> Frame 125: 53 bytes on wire (424 bits), 53 bytes captured (424 bits) on interface \D
> Ethernet II, Src: HP_56:5f:9c (e8:d8:d1:56:5f:9c), Dst: STMicroelect_00:00:00 (00:80
> Internet Protocol Version 4, Src: 169.254.32.8, Dst: 169.254.32.100
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Data (11 bytes)
    Data: 37332e3638323732363839
    [Length: 11]

0000  00 80 e1 00 00 00 e8 d8  d1 56 5f 9c 08 00 45 00   ·········V_···E·
0010  00 27 63 57 00 00 80 11  00 00 a9 fe 20 08 a9 fe   ·'cW···· ········
0020  20 64 c3 50 c3 50 00 13  94 8d 37 33 2e 36 38 32    d·P·P·· ··73.682
0030  37 32 36 38 39                                     72689
```

Figure 4-3: UDP Numerical Data Transmission and ACK (Tera Term, Python and Wireshark)

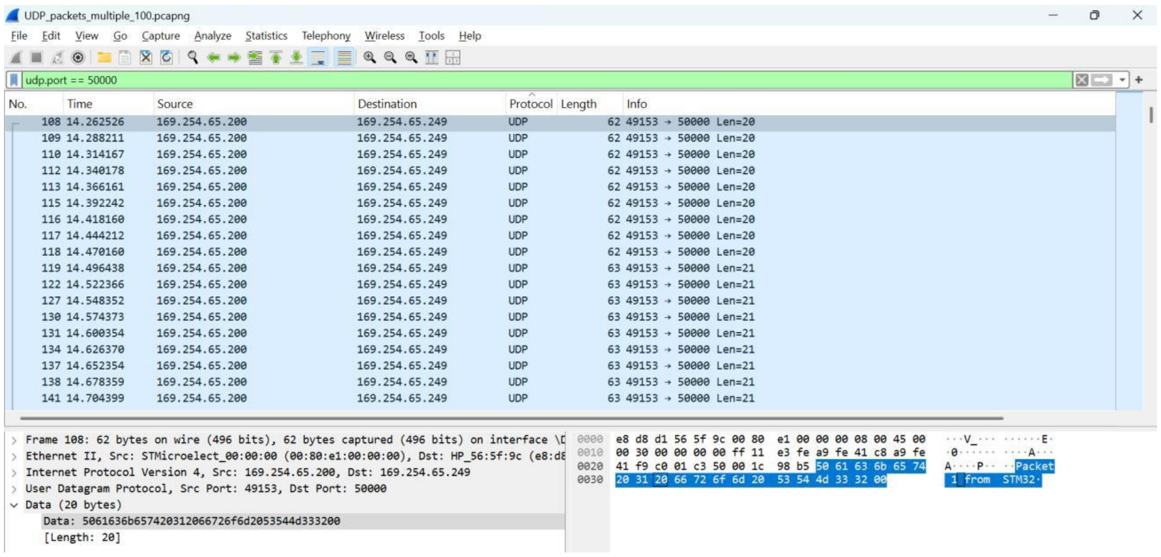
4.1.2 STM32 as a UDP Client: Data Transmission

Apart from functioning as a UDP server, the STM32F207ZG board's capability to act as a UDP client and initiate data transmission was also verified. This step was essential to confirm the potential of the STM32 board to perform bidirectional communication. For this test, firmware was developed to send a burst of formatted string messages, ranging from 100 to infinite in number, from the STM32 board to a designated IP address and port on a host PC. In this scenario, the STM32 board responded in a way where an embedded device might continuously send status updates, logged data, or alerts to a monitoring system.

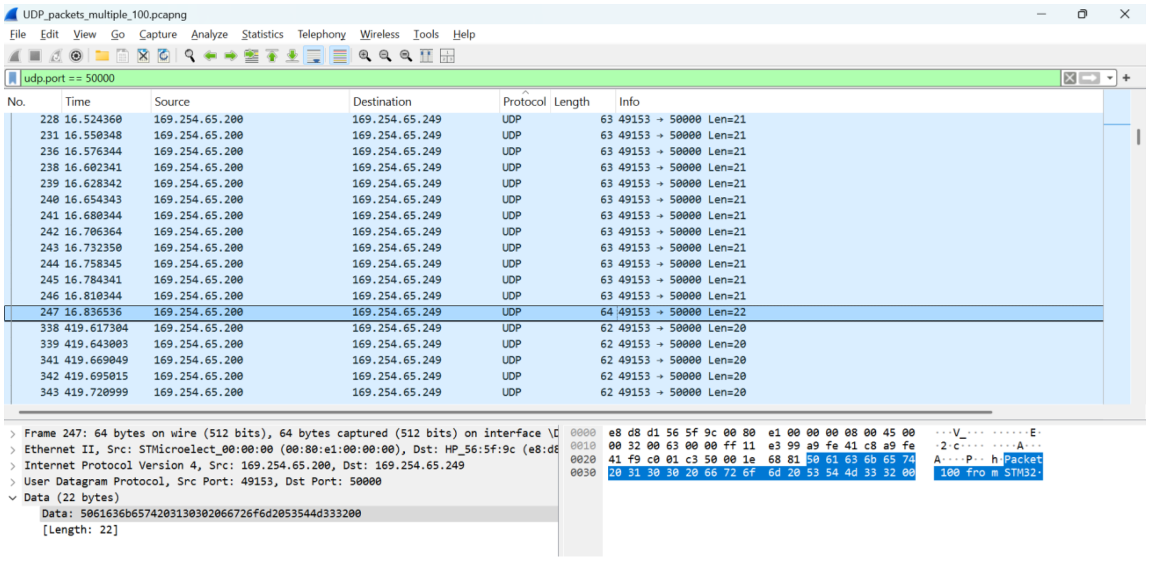
Within the STM32 firmware, important steps were taken to carry out the implementation of the board as a UDP client. A dedicated function named *send_multiple_udp_packets* was created to manage the communication with the client side. Inside this function, a new UDP Protocol Control Block (PCB) was allocated using *udp_new()*. Moreover, instead of binding this PCB to a local port for listening (as was done in the server configuration), it was connected to the target PC's IP address (e.g., 169.254.65.249) and a specific UDP port (e.g., 50000) using *udp_connect()*. By doing so, it established a connection for the outgoing packets that allowed LwIP to handle the necessary routing.

An initial "ARP Trigger Packet" was transmitted on the PBUF_LINK layer right after the *udp_connect()* function to guarantee correct network resolution, particularly with respect to Address Resolution Protocol (ARP) for mapping the target IP to a MAC address. A small delay (e.g., *HAL_Delay(1000)*) was then added that allowed time for any ARP exchanges to complete before starting the main data transmission sequence. Following this, a loop was executed ranging from one hundred times to infinite. In each iteration, a string message, typically "Packet X from STM32," where X was the packet number, was dynamically formatted using sprintf. This message was then encapsulated within a pbuf allocated using *pbuf_alloc(PBUF_TRANSPORT*, ...), and transmitted using *udp_send()*. After each *udp_send()*, a small delay and calls to *ethernetif_input(&gnetif)* and *sys_check_timeouts()* were included to facilitate LwIP's internal processing and actual packet dispatch over the Ethernet link. After completion of the loop, the UDP PCB was disconnected using *udp_disconnect()* and then removed with *udp_remove()* to free up resources.

Successful transmission of these packets was verified on the host PC, which used a Python UDP receiver script and Wireshark to receive these packets and capture the network traffic. The Wireshark logs clearly showed these 100 UDP packets sent from the STM32's IP address and received by the configured PC IP and port that confirmed the STM32's capability to initiate and manage UDP communication.


UDP_packets_multiple_100.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
udp.port == 50000

No.   Time       Source          Destination     Protocol Length Info
  108 14.262526  169.254.65.200  169.254.65.249  UDP        62 49153 → 50000 Len=20
  109 14.288211  169.254.65.200  169.254.65.249  UDP        62 49153 → 50000 Len=20
  110 14.314167  169.254.65.200  169.254.65.249  UDP        62 49153 → 50000 Len=20
  112 14.340178  169.254.65.200  169.254.65.249  UDP        62 49153 → 50000 Len=20
  113 14.366161  169.254.65.200  169.254.65.249  UDP        62 49153 → 50000 Len=20
  115 14.392242  169.254.65.200  169.254.65.249  UDP        62 49153 → 50000 Len=20
  116 14.418160  169.254.65.200  169.254.65.249  UDP        62 49153 → 50000 Len=20
  117 14.444212  169.254.65.200  169.254.65.249  UDP        62 49153 → 50000 Len=20
  118 14.470160  169.254.65.200  169.254.65.249  UDP        62 49153 → 50000 Len=20
  119 14.496438  169.254.65.200  169.254.65.249  UDP        63 49153 → 50000 Len=21
  122 14.522366  169.254.65.200  169.254.65.249  UDP        63 49153 → 50000 Len=21
  127 14.548352  169.254.65.200  169.254.65.249  UDP        63 49153 → 50000 Len=21
  130 14.574373  169.254.65.200  169.254.65.249  UDP        63 49153 → 50000 Len=21
  131 14.600354  169.254.65.200  169.254.65.249  UDP        63 49153 → 50000 Len=21
  134 14.626370  169.254.65.200  169.254.65.249  UDP        63 49153 → 50000 Len=21
  137 14.652354  169.254.65.200  169.254.65.249  UDP        63 49153 → 50000 Len=21
  138 14.678359  169.254.65.200  169.254.65.249  UDP        63 49153 → 50000 Len=21
  141 14.704399  169.254.65.200  169.254.65.249  UDP        63 49153 → 50000 Len=21

> Frame 108: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface \D
> Ethernet II, Src: STMicroelect_00:00:00 (00:80:e1:00:00:00), Dst: HP_56:5f:9c (e8:d8
> Internet Protocol Version 4, Src: 169.254.65.200, Dst: 169.254.65.249
> User Datagram Protocol, Src Port: 49153, Dst Port: 50000
v Data (20 bytes)
    Data: 5061636b657420312066726f6d2053544d333200
    [Length: 20]

0000  e8 d8 d1 56 5f 9c 00 80  e1 00 00 00 08 00 45 00   ···V_··· ······E·
0010  00 30 00 00 00 00 ff 11  e3 fe a9 fe 41 c8 a9 fe   ·0······ ····A···
0020  41 f9 c0 01 c3 50 00 1c  98 b5 50 61 63 6b 65 74   A····P·· ··Packet
0030  20 31 20 66 72 6f 6d 20  53 54 4d 33 32 00          1 from  STM32·
```



```
udp.port == 50000

No.   Time        Source          Destination     Protocol Length Info
  228 16.524360   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  231 16.550348   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  236 16.576344   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  238 16.602341   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  239 16.628342   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  240 16.654343   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  241 16.680344   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  242 16.706364   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  243 16.732350   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  244 16.758345   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  245 16.784341   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  246 16.810344   169.254.65.200  169.254.65.249  UDP      63 49153 → 50000 Len=21
  247 16.836536   169.254.65.200  169.254.65.249  UDP      64 49153 → 50000 Len=22
  338 419.617304  169.254.65.200  169.254.65.249  UDP      62 49153 → 50000 Len=20
  339 419.643003  169.254.65.200  169.254.65.249  UDP      62 49153 → 50000 Len=20
  341 419.669049  169.254.65.200  169.254.65.249  UDP      62 49153 → 50000 Len=20
  342 419.695015  169.254.65.200  169.254.65.249  UDP      62 49153 → 50000 Len=20
  343 419.720999  169.254.65.200  169.254.65.249  UDP      62 49153 → 50000 Len=20

> Frame 247: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on interface \D
> Ethernet II, Src: STMicroelect_00:00:00 (00:80:e1:00:00:00), Dst: HP_56:5f:9c (e8:d8
> Internet Protocol Version 4, Src: 169.254.65.200, Dst: 169.254.65.249
> User Datagram Protocol, Src Port: 49153, Dst Port: 50000
v Data (22 bytes)
    Data: 5061636b65742031303020666726f6d2053544d333200
    [Length: 22]

0000  e8 d8 d1 56 5f 9c 00 80  e1 00 00 00 08 00 45 00   ···V_··· ······E·
0010  00 32 00 63 00 00 ff 11  e3 99 a9 fe 41 c8 a9 fe   ·2·c···· ····A···
0020  41 f9 c0 01 c3 50 00 1e  68 81 50 61 63 6b 65 74   A····P·· h·Packet
0030  20 31 30 30 20 66 72 6f  6d 20 53 54 4d 33 32 00    100 fro m STM32·
```

Figure 4-4: STM32 as UDP Client

4.1.3 LwIP Stack Configuration

The Lightweight IP (LwIP) stack is an open-source TCP/IP suite designed for embedded systems. It is essential for enabling Ethernet communication on the STM32F207ZG microcontroller. Proper configuration of LwIP is necessary for optimizing the performance of the application, as this configuration allows for the proper resource utilization that suits the hardware constraints. Several key parameters were adjusted in the lwipopts.h configuration file so that this project can be operated using a bare-metal (non-RTOS) approach.

The core setting for the bare-metal system was *NO_SYS = 1*, which configured LwIP to run without relying on an operating system. By doing so, network events are processed using the main loop pooling mechanism. Since the focus of this project was solely on UDP communication, TCP support was disabled by setting *LWIP_TCP* to 0. To ensure that the STM32's Ethernet MAC capabilities are fully utilized, tasks such as checksum generation and verification of IP and UDP protocols were offloaded from the CPU, and this was done by setting *CHECKSUM_BY_HARDWARE* to 1.

Memory management was carefully done within the LwIP CubeMX settings. *MEM_ALIGNMENT* was set to 4 bytes, which is standard for 32-bit microcontrollers. The total heap memory available to LwIP (*MEM_SIZE*) was set to a suitable value (e.g., 50KB), as this provided a balance to the needs of network buffers and other LwIP internal structures against the available SRAM. The number of packet buffers (*MEMP_NUM_PBUF* and *PBUF_POOL_SIZE*) and their sizes (*PBUF_POOL_BUFSIZE*, e.g., 1600 bytes) were adjusted to handle the expected UDP packet sizes and traffic volume without excessive memory usage. Especially for UDP, the number of active UDP Protocol Control Blocks (*MEMP_NUM_UDP_PCB*) was set (e.g., to 6) to accommodate the server-client operations. Few other parameters, such as ARP table size

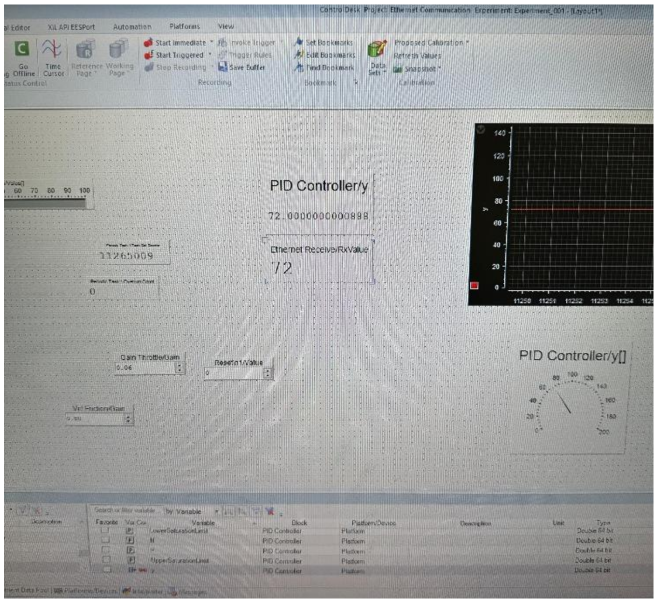
(*ARP_TABLE_SIZE*) and some timeout values, were generally kept at their default values to ensure that there is stable network operation with respect to the HIL simulator or PC communication. Collectively, these configurations provided a functional network stack that can be used for real-time UDP messaging.

4.1.4 Integration with dSPACE SCALEXIO for UDP Echo Testing

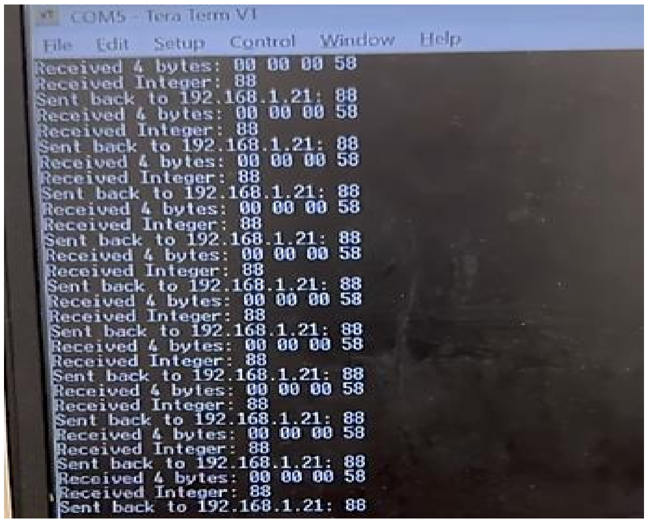
An important milestone in the Ethernet phase was the integration and testing of UDP communication between the STM32F207ZG board and the dSPACE SCALEXIO HIL simulator. This step was crucial as it verified the network connectivity and data exchange pipeline before implementing the SPI bridging concept. For this test, the STM32 board was configured as a UDP server, just like the firmware described in Section 4.1.1.3, where it was programmed to listen for incoming UDP packets, parse a floating-point number from the payload, and then echo this numerical value back to the sender (Python script).

The dSPACE SCALEXIO system, running a real-time application, acted as the UDP client. The dSPACE SCALEXIO system was configured to send UDP packets containing floating-point numbers to the STM32 Master's IP address (192.168.1.22) and port (44000). The same model running on dSPACE SCALEXIO system was also configured to receive the echoed UDP packets from the STM32 on IP address (192.168.1.21) and port (e.g., 44001).

The successful execution of this test was verified by observing the data flow in real-time. As shown in the Figure 4-5: UDP Communication with SCALEXIO Simulator, the data values sent from the dSPACE ControlDesk interface via a slider named "PID Controller/y" were received by the STM32 board. In response to this, the STM32 board processed and displayed these values on its serial terminal (Tera Term). Subsequently, it echoed the same values back to the dSPACE real-time application, which monitored these values within the dSPACE ControlDesk via a display screen, "Ethernet Receive/RxValue" and a graph. This end-to-end test successfully demonstrated basic real-time UDP connectivity and data exchange between the embedded STM32 platform and the dSPACE HIL simulation environment, which was a breakthrough as it paved the way for more complex system integration.



PID Controller/y
72.000000000888
Ethernet Receive/RxValue
72
Periodic Task 1/Overrun Count
0
Gain Throttle/Gain
0.06
Resetin1/Value
0



```
COM5 - Tera Term VT
File  Edit  Setup  Control  Window  Help
Received 4 bytes: 00 00 00 58
Received Integer: 88
Sent back to 192.168.1.21: 88
Received 4 bytes: 00 00 00 58
Received Integer: 88
Sent back to 192.168.1.21: 88
Received 4 bytes: 00 00 00 58
Received Integer: 88
Sent back to 192.168.1.21: 88
Received 4 bytes: 00 00 00 58
Received Integer: 88
Sent back to 192.168.1.21: 88
Received 4 bytes: 00 00 00 58
Received Integer: 88
Sent back to 192.168.1.21: 88
Received 4 bytes: 00 00 00 58
Received Integer: 88
Sent back to 192.168.1.21: 88
Received 4 bytes: 00 00 00 58
Received Integer: 88
Sent back to 192.168.1.21: 88
Received 4 bytes: 00 00 00 58
Received Integer: 88
Sent back to 192.168.1.21: 88
Received 4 bytes: 00 00 00 58
Received Integer: 88
Sent back to 192.168.1.21: 88
Received 4 bytes: 00 00 00 58
Received Integer: 88
Sent back to 192.168.1.21: 88
```

Figure 4-5: UDP Communication with SCALEXIO Simulator

4.2 SPI Phase: Inter-STM32 Board Communication

After successfully establishing the Ethernet UDP functionality on a single board, the next target was to develop a robust SPI communication link between two STM32F207ZG boards, where one was considered the Master and the other the Slave. This phase was important for creating a reliable, high-speed, full-duplex data transfer mechanism using Direct Memory Access (DMA), which would form the core data pipe for the sensor data emulation and the Ethernet-SPI bridging. The development progressed through various stages, starting from a basic data exchange and then step by step adding synchronization, framing, error detection, and more complex data handling capabilities.

4.2.1 Basic SPI DMA Configuration and Full-Duplex Transfer

The initial setup involved configuring the SPI1 peripheral on both STM32 boards for communication. The Master board's SPI1 peripheral was set to Master mode with hardware NSS output functionality enabled on its designated NSS pin (PA4). Subsequently, the Slave board's SPI1 peripheral was set to Slave mode with hardware NSS input on its designated NSS pin (PA4). This ensured that the Slave device would be automatically selected and deselected by the Master's NSS signal. Both peripherals were set up for 8-bit data size, CPOL 0, and CPHA 1Edge (SPI Mode 0). The APB2 peripheral clock was 48 MHz; hence, selecting the baud rate prescaler as 32 resulted in an SPI clock speed of 1.5 Mbits/sec ((48 MHz/32) = 1.5 Mbits/sec). Full-duplex communication was carried out by employing DMA for both transmission and reception on each board, utilizing *HAL_SPI_TransmitReceive_DMA()*.

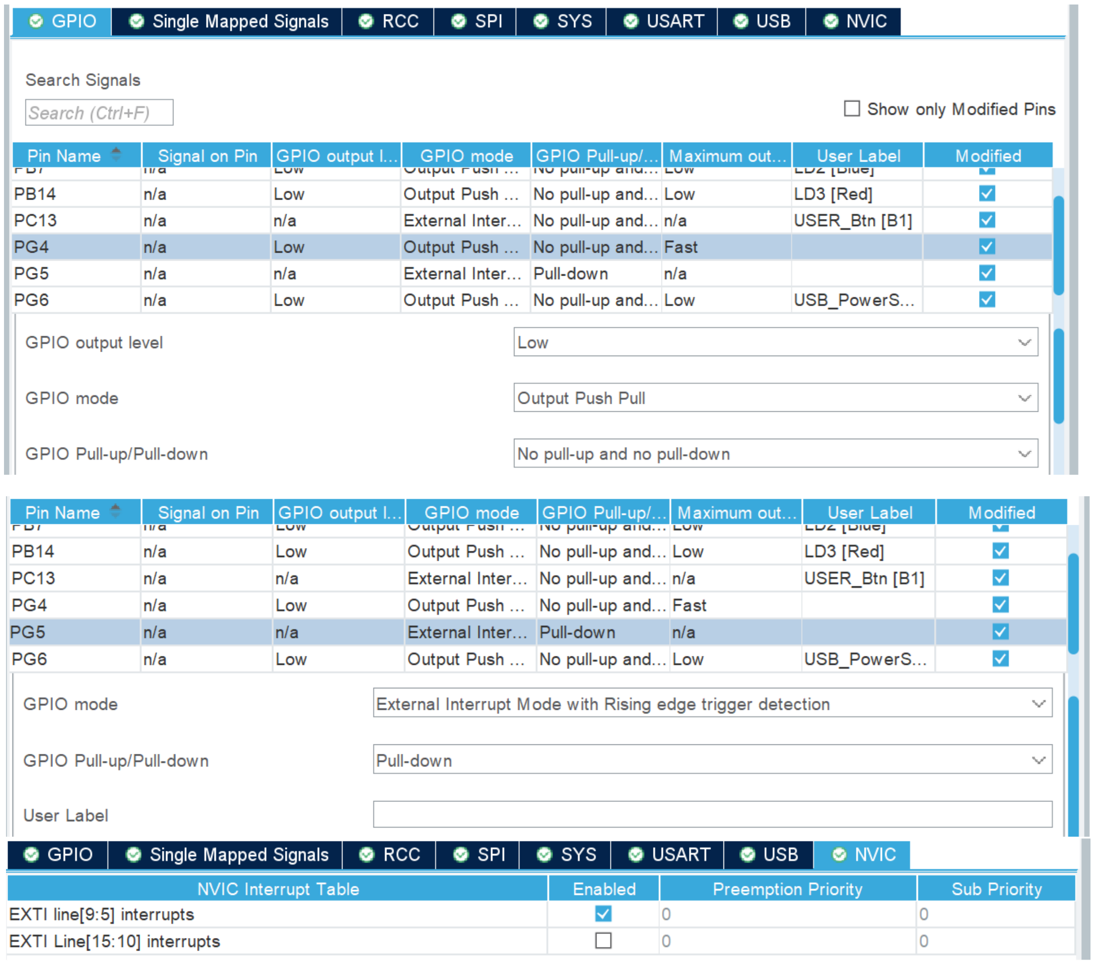
Table 4-1: SPI Full-Duplex Pin configuration

Function	**STM32 Pin**	**Nucleo board Pin Name**	**Connector**	**Pin Number**
SCK	PA5	SPI_A_SCK	CN7	D13 (Pin 10)
MISO	PA6	SPI_A_MISO	CN7	D12 (Pin 12)
MOSI	PB5	SPI_A_MOSI	CN7	D11 (Pin 14)
NSS	PA4	SPI_B_NSS	CN7	D24 (Pin 17)

The initial testing involved the Master board sending a fixed string, “Hello SPI,” to the Slave board at an interval of 1 second using the *HAL_Delay(1000)* command. The slave board was programmed to echo back the same string to the master board. The results were verified visually on the serial terminals (Tera term) of both Master and Slave boards. These early tests showed successful basic data exchange using SPI with DMA. However, there were some challenges related to synchronization when the code was run for a longer period of time and when the data sending rate was increased (for example, from a 1-second interval to a 100-ms interval). This indicated that though the basic data transfer was functional, the Slave could halt or miss data without a more robust timing and proper handshake, especially in the case where the Master initiated transactions when the Slave was not ready.

4.2.2 GPIO and Timer-Based Synchronization for 1ms SPI Transactions

To address the synchronization issue and perform a reliable SPI transaction at a faster rate (e.g., 1 ms), a GPIO-based handshake mechanism was introduced. GPIO in PG5 of the Master board was set as an EXTI (external interrupt rising edge triggered) in pull-down configuration to detect the “ready” signal from the Slave. GPIO pin PG4 on the Master board was configured as output push-pull to send a “master ready” signal to the slave. Similarly, PG4 of Slave board was configured as an input (with pull-down) to detect the “master ready” signal from Master board, and PG5 was configured as an output push-pull to send the “ready” signal to the Master board. The Slave, upon detecting the “master ready” signal, would prepare its response and then pulse its PG5 for a short period, which triggers the EXTI on the Mater. The Master’s EXTI service routine would then initiate the *HAL_SPI_TransmitReceive_DMA()* operation.



Pin Name	Signal on Pin	GPIO output l...	GPIO mode	GPIO Pull-up/...	Maximum out...	User Label	Modified
PB14	n/a	Low	Output Push ...	No pull-up and...	Low	LD3 [Red]	✓
PC13	n/a	n/a	External Inter...	No pull-up and...	n/a	USER_Btn [B1]	✓
PG4	n/a	Low	Output Push ...	No pull-up and...	Fast		✓
PG5	n/a	n/a	External Inter...	Pull-down	n/a		✓
PG6	n/a	Low	Output Push ...	No pull-up and...	Low	USB_PowerS...	✓

GPIO output level: Low

GPIO mode: Output Push Pull

GPIO Pull-up/Pull-down: No pull-up and no pull-down

Pin Name	Signal on Pin	GPIO output l...	GPIO mode	GPIO Pull-up/...	Maximum out...	User Label	Modified
PB14	n/a	Low	Output Push ...	No pull-up and...	Low	LD3 [Red]	✓
PC13	n/a	n/a	External Inter...	No pull-up and...	n/a	USER_Btn [B1]	✓
PG4	n/a	Low	Output Push ...	No pull-up and...	Fast		✓
PG5	n/a	n/a	External Inter...	Pull-down	n/a		✓
PG6	n/a	Low	Output Push ...	No pull-up and...	Low	USB_PowerS...	✓

GPIO mode: External Interrupt Mode with Rising edge trigger detection

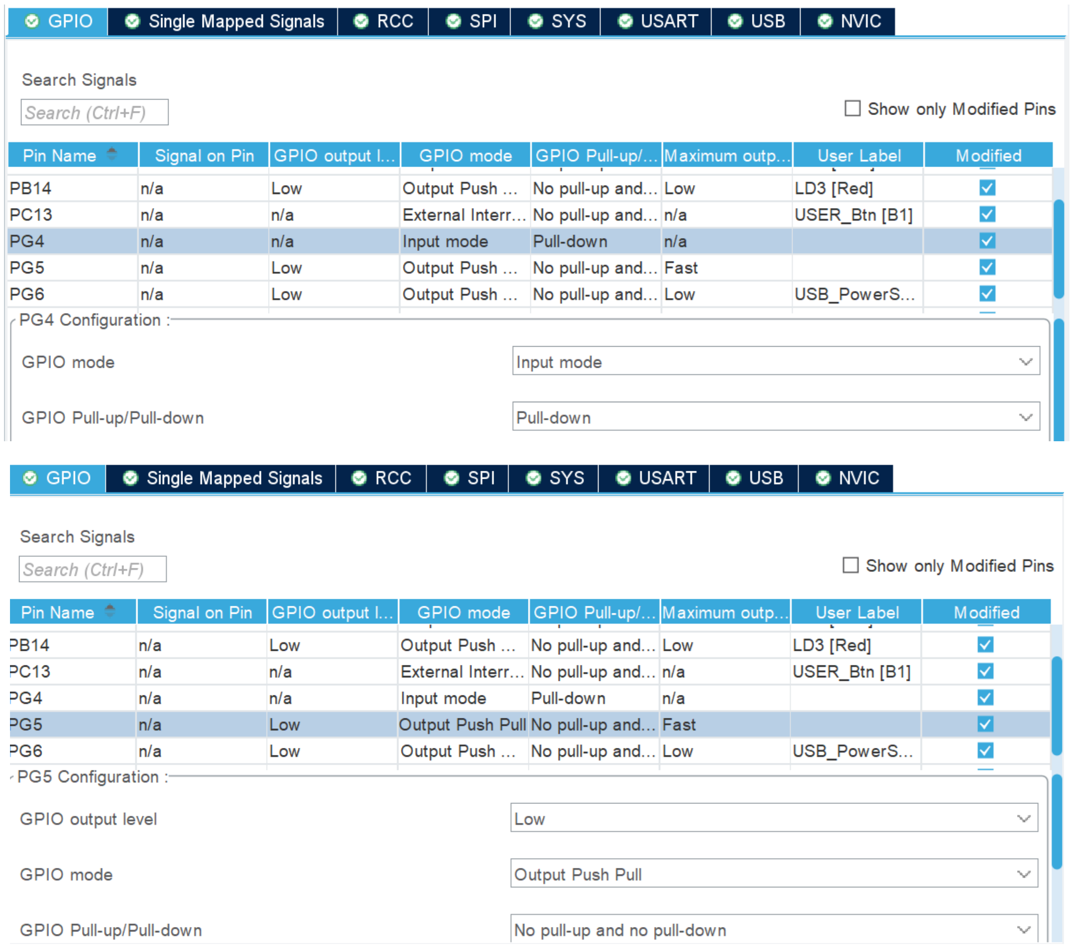
GPIO Pull-up/Pull-down: Pull-down

User Label:

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
EXTI line[9:5] interrupts	✓	0	0
EXTI Line[15:10] interrupts	☐	0	0

Figure 4-6: Master Board SPI Handshake GPIO Configuration

The Slave board’s Handshake GPIO configuration is as shown in Figure 4-7: Slave Board SPI Handshake GPIO Configuration.



Pin Name	Signal on Pin	GPIO output l...	GPIO mode	GPIO Pull-up/...	Maximum outp...	User Label	Modified
PB14	n/a	Low	Output Push ...	No pull-up and...	Low	LD3 [Red]	✓
PC13	n/a	n/a	External Interr...	No pull-up and...	n/a	USER_Btn [B1]	✓
PG4	n/a	n/a	Input mode	Pull-down	n/a		✓
PG5	n/a	Low	Output Push ...	No pull-up and...	Fast		✓
PG6	n/a	Low	Output Push ...	No pull-up and...	Low	USB_PowerS...	✓

PG4 Configuration :

GPIO mode: Input mode

GPIO Pull-up/Pull-down: Pull-down

Pin Name	Signal on Pin	GPIO output l...	GPIO mode	GPIO Pull-up/...	Maximum outp...	User Label	Modified
PB14	n/a	Low	Output Push ...	No pull-up and...	Low	LD3 [Red]	✓
PC13	n/a	n/a	External Interr...	No pull-up and...	n/a	USER_Btn [B1]	✓
PG4	n/a	n/a	Input mode	Pull-down	n/a		✓
PG5	n/a	Low	Output Push Pull	No pull-up and...	Fast		✓
PG6	n/a	Low	Output Push ...	No pull-up and...	Low	USB_PowerS...	✓

PG5 Configuration :

GPIO output level: Low

GPIO mode: Output Push Pull

GPIO Pull-up/Pull-down: No pull-up and no pull-down

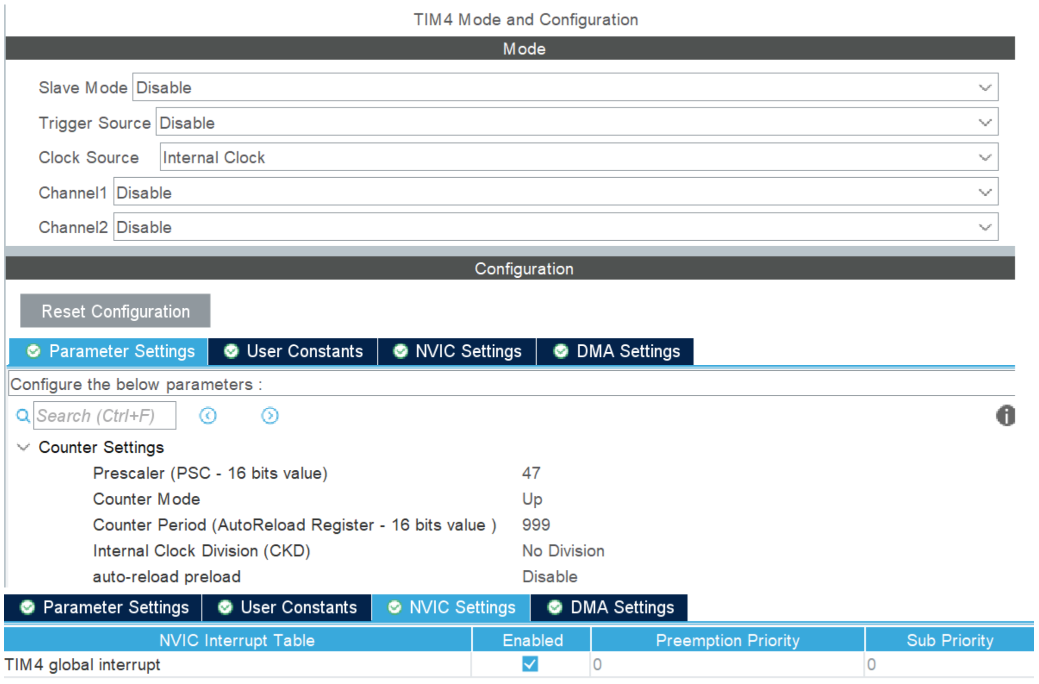
Figure 4-7: Slave Board SPI Handshake GPIO Configuration

While the EXTI-based GPIO handshake improved synchronization and responsiveness, further refinement were done for consistent 1-ms polling by the Master. This led to the implementation of a hardware timer, and TIM4 was selected as it is a 16-bit general purpose timer, well suited for this application. TIM4 on the Master board was configured to generate an interrupt every 1-ms and the calculations for selecting an appropriate prescaler (PSC) and counter period (auto-reload register (ARR)) value was carried out using the following formula.

While the EXTI-based GPIO handshake approach improved synchronization and responsiveness, further enhancement was done for consistent 1-ms polling by the Master. This led to the implementation of a hardware timer, and TIM4 was selected as it is a 16-bit general-purpose timer, well suited for this application. TIM4 on the Master board was configured to generate an interrupt every 1-ms and the calculations for selecting an appropriate prescaler (PSC) and counter period (auto-reload register (ARR)) value were carried out using the following formula.

$$Update\ Frequency = \frac{TIM\ Clock}{PSC + (ARR + 1)}$$

Timer TIM4 is linked with the APB1 clock, which is 48 MHz; hence, selecting PSC as 47 and ARR as 999 resulted in Update Frequency = 1000 Hz, which is 1-ms. The *HAL_TIM_PeriodElapsedCallback* function then became responsible for initiating the SPI DMA transaction. It provided the Slave indicated readiness (e.g., by the Master checking the state of its PG5 input, which was connected to the Slave's PG5 output). The Slave board, in this configuration, continued to use its PG5 pin as an output to signal when it was ready for a transaction, and it monitored the Master's signal “master ready” (PG4 from Master connected to Slave's PG4) for SPI transaction initiation. This timer-driven approach on the Master along with the "slave ready" signal, provided a more deterministic 1-ms SPI polling interval.



Slave Mode: Disable
Trigger Source: Disable
Clock Source: Internal Clock
Channel1: Disable
Channel2: Disable

Counter Settings

Prescaler (PSC - 16 bits value)	47
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits value)	999
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM4 global interrupt	✓	0	0

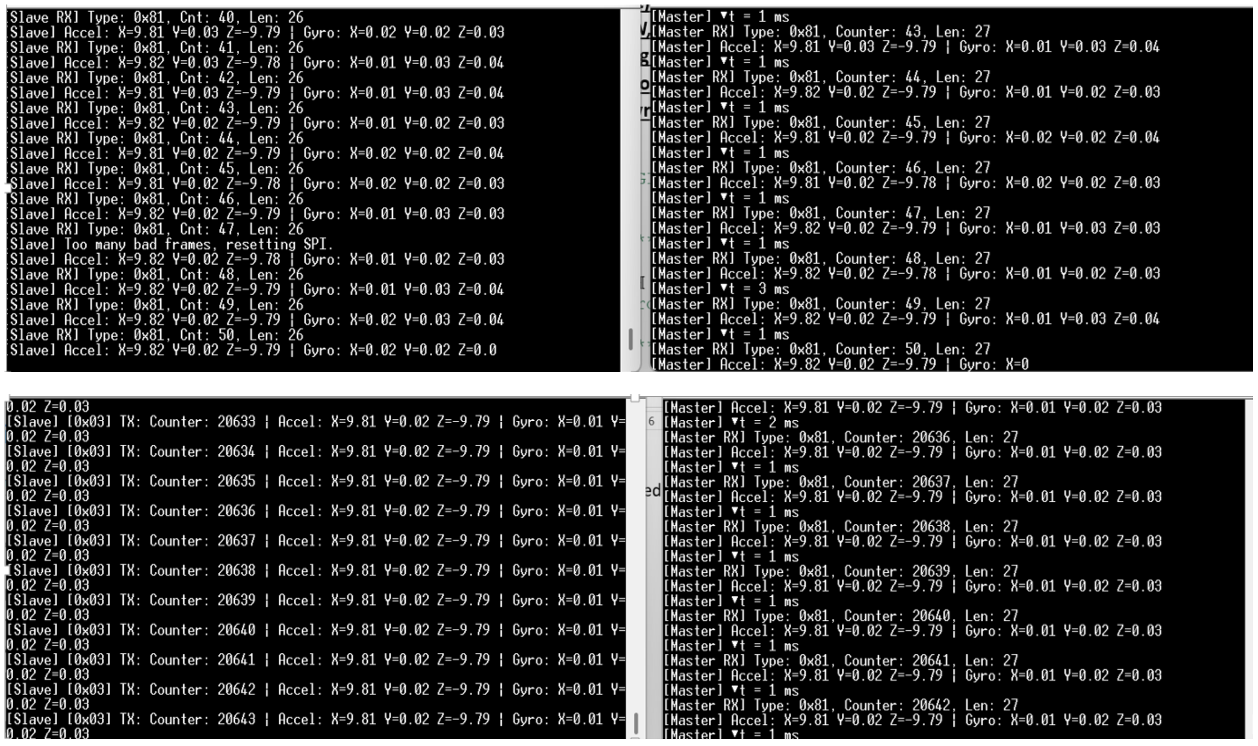
Figure 4-8: Master Board Timer (TIM4) Configuration for SPI

4.2.3 Implementation of Custom SPI Framing and CRC-8 for Data Integrity

To enhance robustness and support variable data types and lengths, a customized SPI framing protocol was implemented on Master and Slave communication. Each frame (as detailed in Section 3.2.3 and depicted in Figure 3-6: Custom SPI Frame Structure) consisted of a *START_BYTE (0xAA)*, a 1-byte *LENGTH* field (indicating the length of *TYPE* + *PAYLOAD*), a 1-byte *TYPE* field, a variable-length *PAYLOAD*, a software-calculated 8-bit Cyclic Redundancy Check (CRC-8) that covered the *TYPE* and *PAYLOAD*, and an *END_BYTE (0x55)*. The *build_frame* function was developed to construct these frames, and subsequent parsing logic was implemented on the receiving side to validate the frame structure and extract the payload. The CRC-8 implementation was done using a polynomial (0x07) that provided a mechanism for transmission error detection over the SPI link. All SPI DMA transactions were set to transfer a fixed maximum frame size (e.g., 32 bytes), whereas the actual frame content and length were determined by the parsing logic based on the frame's *LENGTH* field.

4.2.4 Slave Sensor Emulation and Packet Integrity Checks

The firmware of the slave STM32 board was enhanced to emulate the behavior of a sensor (e.g., an inertial measurement unit (IMU)). Initially, this was implemented as Slave sending back static framed data but was later upgraded to include variable data types (e.g., int16_t, float, uint8_t) and randomly generated values (e.g., IMU's accelerometer and gyroscope reading) for these data types to simulate a real sensor's output. A packet counter (*packetCbadFrameCountounter* on the Slave, echoed and checked as *expectedCounter* on the Master) was incorporated within the payload of data frames. This allowed the Master to detect lost packets by comparing the received counter value with its expected sequence. If a mismatch occurred, the number of lost packets was calculated and displayed on the serial display of Master. Furthermore, robust error-handling mechanisms, which included SPI timeout detection (*SPI_TIMEOUT_MS, lastRxTick*) and a tracker, were implemented on both Master and Slave boards. If multiple consecutive invalid frames were received or if an SPI transaction timed out, the respective board would attempt to re-initialize its SPI peripheral to recover the communication link.



```
Slave RX] Type: 0x81, Cnt: 40, Len: 26
Slave] Accel: X=9.81 Y=0.03 Z=-9.79 ¦ Gyro: X=0.02 Y=0.02 Z=0.03
Slave RX] Type: 0x81, Cnt: 41, Len: 26
Slave] Accel: X=9.82 Y=0.03 Z=-9.78 ¦ Gyro: X=0.01 Y=0.03 Z=0.04
Slave RX] Type: 0x81, Cnt: 42, Len: 26
Slave] Accel: X=9.81 Y=0.03 Z=-9.79 ¦ Gyro: X=0.01 Y=0.03 Z=0.04
Slave RX] Type: 0x81, Cnt: 43, Len: 26
Slave] Accel: X=9.82 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
Slave RX] Type: 0x81, Cnt: 44, Len: 26
Slave] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.02 Y=0.02 Z=0.04
Slave RX] Type: 0x81, Cnt: 45, Len: 26
Slave] Accel: X=9.81 Y=0.02 Z=-9.78 ¦ Gyro: X=0.02 Y=0.02 Z=0.03
Slave RX] Type: 0x81, Cnt: 46, Len: 26
Slave] Accel: X=9.82 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.03 Z=0.03
Slave RX] Type: 0x81, Cnt: 47, Len: 26
Slave] Too many bad frames, resetting SPI.
Slave] Accel: X=9.82 Y=0.02 Z=-9.78 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
Slave RX] Type: 0x81, Cnt: 48, Len: 26
Slave] Accel: X=9.82 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.03 Z=0.04
Slave RX] Type: 0x81, Cnt: 49, Len: 26
Slave] Accel: X=9.82 Y=0.02 Z=-9.79 ¦ Gyro: X=0.02 Y=0.03 Z=0.04
Slave RX] Type: 0x81, Cnt: 50, Len: 26
Slave] Accel: X=9.82 Y=0.02 Z=-9.79 ¦ Gyro: X=0.02 Y=0.02 Z=0.0
```

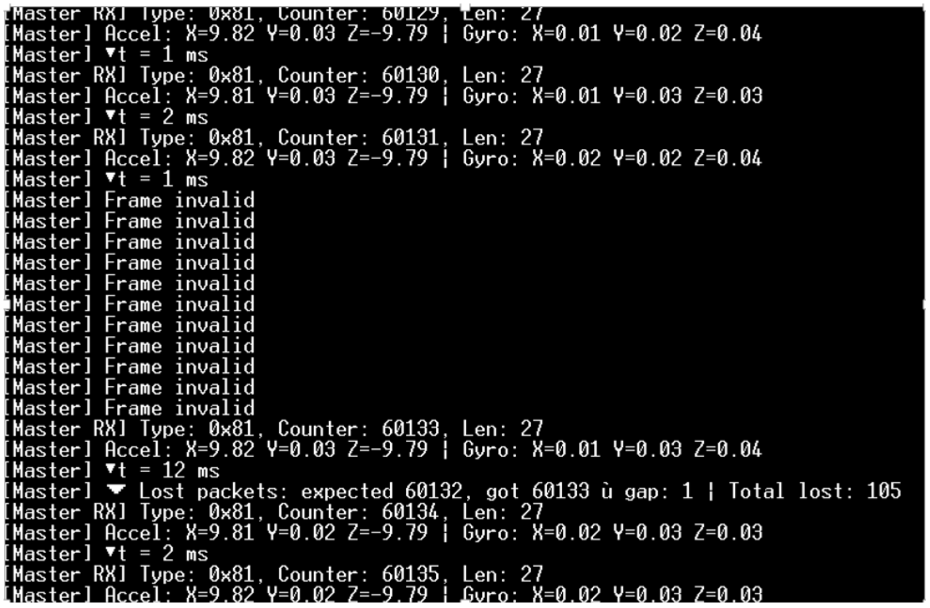
```
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 43, Len: 27
[Master] Accel: X=9.81 Y=0.03 Z=-9.79 ¦ Gyro: X=0.01 Y=0.03 Z=0.04
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 44, Len: 27
[Master] Accel: X=9.82 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 45, Len: 27
[Master] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.02 Y=0.02 Z=0.04
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 46, Len: 27
[Master] Accel: X=9.81 Y=0.02 Z=-9.78 ¦ Gyro: X=0.02 Y=0.02 Z=0.03
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 47, Len: 27
[Master] Accel: X=9.82 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.03 Z=0.03
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 48, Len: 27
[Master] Accel: X=9.82 Y=0.02 Z=-9.78 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
[Master] ▼t = 3 ms
[Master RX] Type: 0x81, Counter: 49, Len: 27
[Master] Accel: X=9.82 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.03 Z=0.04
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 50, Len: 27
[Master] Accel: X=9.82 Y=0.02 Z=-9.79 ¦ Gyro: X=0
```

```
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20633 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20634 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20635 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20636 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20637 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20638 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20639 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20640 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20641 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20642 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
[Slave] [0x03] TX: Counter: 20643 ¦ Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=
0.02 Z=0.03
```

```
[Master] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
[Master] ▼t = 2 ms
[Master RX] Type: 0x81, Counter: 20636, Len: 27
[Master] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 20637, Len: 27
[Master] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 20638, Len: 27
[Master] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 20639, Len: 27
[Master] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 20640, Len: 27
[Master] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 20641, Len: 27
[Master] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
[Master] ▼t = 1 ms
[Master RX] Type: 0x81, Counter: 20642, Len: 27
[Master] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.03
[Master] ▼t = 1 ms
```

Figure 4-9: Master and Slave Full-Duplex Communication - Sensor Data

The Figure 4-10: Master board's UART output showing detected SPI packet loss shows total packets lost and packets detected.



```
[Master RX] Type: 0x81, Counter: 60129, Len: 27
[Master] Accel: X=9.82 Y=0.03 Z=-9.79 ¦ Gyro: X=0.01 Y=0.02 Z=0.04
[Master] ▾t = 1 ms
[Master RX] Type: 0x81, Counter: 60130, Len: 27
[Master] Accel: X=9.81 Y=0.03 Z=-9.79 ¦ Gyro: X=0.01 Y=0.03 Z=0.03
[Master] ▾t = 2 ms
[Master RX] Type: 0x81, Counter: 60131, Len: 27
[Master] Accel: X=9.82 Y=0.03 Z=-9.79 ¦ Gyro: X=0.02 Y=0.02 Z=0.04
[Master] ▾t = 1 ms
[Master] Frame invalid
[Master] Frame invalid
[Master] Frame invalid
[Master] Frame invalid
[Master] Frame invalid
[Master] Frame invalid
[Master] Frame invalid
[Master] Frame invalid
[Master] Frame invalid
[Master] Frame invalid
[Master] Frame invalid
[Master RX] Type: 0x81, Counter: 60133, Len: 27
[Master] Accel: X=9.82 Y=0.03 Z=-9.79 ¦ Gyro: X=0.01 Y=0.03 Z=0.04
[Master] ▾t = 12 ms
[Master] ▼ Lost packets: expected 60132, got 60133 ù gap: 1 ¦ Total lost: 105
[Master RX] Type: 0x81, Counter: 60134, Len: 27
[Master] Accel: X=9.81 Y=0.02 Z=-9.79 ¦ Gyro: X=0.02 Y=0.03 Z=0.03
[Master] ▾t = 2 ms
[Master RX] Type: 0x81, Counter: 60135, Len: 27
[Master] Accel: X=9.82 Y=0.02 Z=-9.79 ¦ Gyro: X=0.02 Y=0.03 Z=0.03
```

Figure 4-10: Master board's UART output showing detected SPI packet loss

4.3 Ethernet-SPI Bridge Implementation and Prototyping

This is the critical integration phase where the previously validated Ethernet UDP interface (Section 4.1) and the refined SPI communication link (Section 4.2) were combined. The Master board was developed to act as a bridge between Ethernet and SPI link. This integration phase was termed the prototype system, where a Python script, complimented with a Tkinter-based Graphical User Interface (GUI), was used to mimic the role of the dSPACE HIL simulator. The script was used to send UDP-based commands to the Master STM32, which then forwarded these commands to the Slave STM32 via SPI. Slave, which is emulating a multi-functional sensor, processed these commands and returned a response (i.e., ACK/NACK frames, randomly generated speed, and IMU's simulated gyro and acceleration value). This SPI-based response was then received by the Master, which forwarded it as a UDP packet to the Python application for display and analysis. This prototype allowed end-to-end testing of the entire communication pipeline and running logic before transitioning it to the actual dSPACE simulator.

4.3.1 Master Board: Bridging Logic and State Management

The firmware for the Master STM32 board was designed to manage the bidirectional flow of data and commands between the Ethernet (UDP) and SPI interfaces. A major aspect of this design was the implementation of robust state management to handle the running network and peripheral operations.

After getting initialized, the Master board configured its LwIP stack and started a UDP server to listen to the incoming commands from the Python application on a predefined IP address and port.

The *udp_receive_master_command_callback* function was responsible for parsing these incoming UDP packets. These UDP packets contained a 1-byte sub-command from the Python GUI, dictating the Slave to send the desired data stream (e.g., stream idle, speed, gyro, or accelerometer data). Once such a command is received, the callback function updates a global variable named *target_sub_command_for_slave* and sets a flag, *new_control_command_to_send_to_slave*, to signal the main loop that a new control directive was pending for the Slave.

The Master's main loop included logic for managing SPI transactions that was driven by a polling interval (*SPI_POLL_INTERVAL_MS* set as 1-ms and managed using *HAL_GetTick()*). When the SPI interface was idle (*current_spi_state == SPI_STATE_IDLE*) and the polling interval had elapsed, the Master would prepare an SPI transaction. If the *new_control_command_to_send_to_slave* flag was set, the Master would construct an SPI frame of *FRAME_TYPE_CMD_CONTROL_SLAVE*, incorporating the *target_sub_command_for_slave* as its payload. This allowed the Python GUI to actively change the Slave's behavior. If no new control command was pending but a data stream was expected from the Slave (i.e., *target_sub_command_for_slave != SUB_CMD_SET_STREAM_IDLE*), the Master would send a generic "poll" command (a *FRAME_TYPE_CMD_CONTROL_SLAVE* frame with a special payload like 0xFF) to evoke the Slave for data. Before initiating each SPI DMA transfer (*HAL_SPI_TransmitReceive_DMA*), the Master asserted the software-controlled NSS line (GPIO PG4) connected to the Slave's hardware NSS input.

Once *HAL_SPI_TxRxCpltCallback* signals the completion of an SPI transaction, the Master de-asserts the NSS line immediately. The received SPI frame from the Slave (*spi_rx_frame_buf*) is then parsed using the *parse_spi_frame* function, which validates the frame's structure (start/end bytes) and data integrity (CRC-8 checksum). If the frame was valid, its *TYPE field* was analyzed. For responses such as *FRAME_TYPE_SLAVE_ACK* or *FRAME_TYPE_SLAVE_NACK*, the Master updates its internal state (e.g., configAcked) to indicate that the Slave has acknowledged the control command. If the frame contained simulated sensor data (e.g., *FRAME_TYPE_DATA_VEHICLE_SPEED, FRAME_TYPE_DATA_IMU_GYRO, FRAME_ TYPE_DATA_IMU_ACCEL*), the Master then extracted the payload (which included the emulated sensor values and probably a sequence counter from the Slave). This payload, prefixed by the Slave's original SPI frame type byte for context, was then packaged into a new UDP packet and transmitted back to the Python application using the stored IP address and port (*remote_data_dest_ip, remote_data_dest_port*) of the GUI.

The *HAL_SPI_ErrorCallback* function (for hardware-level SPI errors) and a timeout-and-retry mechanism (*check_timeout_and_retry* function) were used to enhance the robustness of the SPI transaction. If the expected acknowledgment or data frame was not received within a defined time (*ACK_TIMEOUT_MS*,) the Master would attempt to resend the last command up to a specified number of times (*MAX_RETRIES*) before it raises the error flag or resets the SPI state, ensuring resilience against transient communication glitches.

4.3.2 Slave Board: Sensor Emulation and Circular DMA Implementation

The Slave STM32 board's firmware was designed to emulate a multi-functional sensor that responds to the SPI commands from the Master and in response to them, it provides continuous data streams. A key challenge in achieving reliable high-frequency (1-ms interval) data exchange was managing the SPI data flow efficiently. Initial implementations that used normal DMA mode for SPI transactions on the Slave encountered issues such as data blackouts, duplicated or very late acknowledgments, and significant packet loss, especially when the firmware was run for a longer period of time. The primary reason for these problems was that in normal DMA, the CPU had to explicitly re-arm the DMA for every single SPI transaction, and if the Master initiated a new transaction before the Slave was ready, data coherency was lost.

In order to counter these limitations, a circular DMA buffering scheme was implemented for both SPI reception (RX) and transmission (TX) on the Slave. Two buffers named *spi_rx_dma_circular_buf* and *spi_tx_dma_circular_buf* were created, and each of them was capable of holding *NUM_BUFFERS_IN_CIRCULAR* (typically 2 for ping-pong buffering) instances of *MAX_FRAME_SIZE*. The DMA controller for the peripheral SPI1 was configured in circular mode in CubeMX. However, this required manual configuration of DMA control register bits, which was beyond the standard CubeMX generation. *CIRC* (circular mode enable) and *MINC* (memory increment enable) and *HTIE* (Half-Transfer Interrupt Enable) and *TCIE* (Transfer Complete Interrupt Enable) which are used for the SPI_RX DMA stream, were set in the "DMA stream x configuration register (DMA_SxCR)." These interrupt settings ensured that the CPU was notified when either the first half or the second half of the circular RX buffer was filled.

SPI communication on the Slave was initiated by the Master asserting the NSS line. The Slave's Hardware NSS pin (PA4) was connected to the Master's GPIO pin (PG4), configured as GPIO output with an initial high level. When the Master asserted this line (driving it LOW), the Slave's SPI peripheral was automatically enabled and allowed it to receive data clocked in by the Master and simultaneously transmit data from its TX buffer.

The data processing and response generation logic of the Slave board was interrupt-driven, as it relied on the DMA transfer complete callback functions that are associated with the circular RX buffer. The STM32 HAL provides distinct callback functions for half-transfer and full-transfer completion in circular DMA mode. The *HAL_SPI_RxHalfCpltCallback(SPI_HandleTypeDef hspi)* function is called by the DMA controller when the first half of the *spi_rx_dma_circular_buf* has been filled with the data received from the Master. Similarly, the *HAL_SPI_RxCpltCallback(SPI_HandleTypeDef hspi)* function is called when the DMA has filled the second half of the buffer. This marks the completion of a full circular pass over that particular segment.

Both of these RX completion callback functions invoke a common processing function called *process_received_data_and_prepare_tx*. This function is responsible for accessing the data within the completed segment of the *spi_rx_dma_circular_buf*. This initial step involves analyzing the received SPI frame by using the function called *parse_spi_function*. This function validates the frame structure by checking the *START_BYTE* and *END_BYTE* fields, verifies the frame's *LENGTH* field, and confirms the data integrity by using a software-implemented CRC-8 checksum.

A 1-byte sub-command from the Master's payload was extracted if a valid frame was received and identified as a *FRAME_TYPE_CMD_CONTROL_SLAVE*. This sub-command (e.g., IDLE, Speed, Gyroscope, or Accelerometer) was then used to update the Slave's internal state variable called *current_data_stream_type_to_send*. In response to such a control command, the Slave firmware then constructs an acknowledgment frame, *FRAME_TYPE_SLAVE_ACK* (or *FRAME_TYPE_SLAVE_NACK* if the sub-command was valid but could not be acted upon for some reason), and echoes back the sub-command and a status byte (e.g., 0x01 for OK, 0x00 for FAIL) as its payload.

If the incoming frame from the Master was a generic poll (e.g., a *FRAME_TYPE_CMD_CONTROL_SLAVE* with a special 0xFF sub-command payload, which the Slave interprets as a request for sending the current data rather than a mode change) or any other setting (non-mode) but validly framed SPI packet, the Slave would proceed to generate sensor data based on the *current_data_stream_type_to_send* state. This involved creating emulated vehicle speed (as a float), IMU gyroscope data (as three int16_t values), or IMU accelerometer data (as three float values). These sensor values were generated with slight random variations around base figures to simulate real sensors and also included an incrementing packet counter for diagnostic purposes by the Master. This emulated sensor data was then framed with its corresponding frame data type identifier (*FRAME_TYPE_DATA_VEHICLE_SPEED, FRAME_TYPE_DATA_IMU_GYRO or FRAME_TYPE_DATA_IMU_ACCEL*) and a new CRC-8.

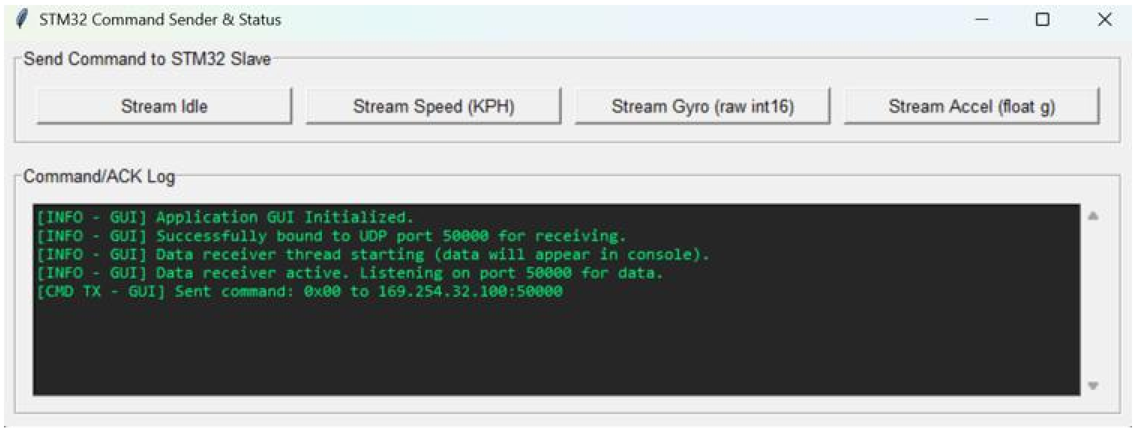
The fully constructed response frame, whether it is an ACK/NACK or a sensor's data frame, is immediately written into the corresponding half of the *spi_tx_dma_circular_buf*. Since the *SPI1_TX DMA* stream was also configured in circular mode and had been initiated once at startup via calling the single *HAL_SPI_TransmitReceive_DMA()*, this prepared data in the TX buffer segment would be automatically clocked out by the Master during the ongoing or subsequent SPI clock cycles, typically at the same time while the RX DMA was filling the other half of its circular buffer.

The *HAL_SPI_ErrorCallback* function was implemented to handle hardware-level SPI errors. After detection of such errors, this callback function re-initializes the SPI peripheral and its related DMA streams by calling *HAL_SPI_Abort()*, *HAL_SPI_DeInit()*, and *MX_SPI1_Init()*, and then it prepares the circular DMA buffers with default ACK frames. Moreover, the SPI re-initialization would also be triggered if the *badFrameCount* logic (which increments upon receiving a corrupt frame) exceeds the threshold of consecutive errors. Implementation of circular DMA and the interrupt management mechanism facilitated achieving continuous high-speed and reliable data exchange in the Ethernet-SPI bridge prototype.

4.3.3 Python GUI for Command and Control

To emulate the command interface of a HIL simulator, a Python application was developed that utilized the Tkinter library for its GUI. This user-friendly application enabled the user to send specific 1-byte UDP-based sub-commands to the Master STM32, hence allowing it to control the data stream requested from the emulated Slave sensor. The GUI had specific buttons, each corresponding to a different sensor mode (e.g., "Stream Idle," "Stream Speed," "Stream Gyro," "Stream Accel"). Clicking a button makes the Python script send the associated sub-command (e.g., 0x00, 0x01, 0x02, 0x03) in a UDP packet to the Master STM32.

The Python script also implemented a non-blocking UDP receiver in a separate thread. This receiver specifically listened only for the UDP packets sent back from the Master STM32. Upon receiving a packet, it parsed the first byte to determine the frame type using *SlaveFrameType* (e.g., ACK/NACK, Speed, Gyro, Accel) and then unpacked the corresponding payload as per the expected format for that frame type. It could unpack status bytes for ACK/NACKs, a 4-byte float for speed, three 2-byte integers for raw gyroscope data, or three 4-byte floats for accelerometer data. The received and parsed information, along with a calculated Round-Trip Time (RTT) for acknowledged commands or data, was displayed in a text log area within the GUI and also printed to the PowerShell console. This setup demonstrated how an operator might interact with the system via a dSPACE ControlDesk interface that allowed sending commands and real-time observation of the system's responses.



```
[INFO - GUI] Application GUI Initialized.
[INFO - GUI] Successfully bound to UDP port 50000 for receiving.
[INFO - GUI] Data receiver thread starting (data will appear in console).
[INFO - GUI] Data receiver active. Listening on port 50000 for data.
[CMD TX - GUI] Sent command: 0x00 to 169.254.32.100:50000
```

Figure 4-11: Python Tkinter GUI for Sending Commands and Displaying Responses

4.3.4 Performance and Robustness Observations of the Prototype System

Extensive testing was done using the Python GUI to send commands and receive data through the Ethernet-SPI Bridge. The system's performance was evaluated based on Round-Trip Time (RTT) for acknowledged data and overall data integrity (packet loss). Initial implementations done using normal DMA on the Slave and simpler polling or EXTI-based synchronization mechanisms on the Master did not show fruitful results, i.e., data blackout periods, reception of duplicated or delayed acknowledgments, and notable packet loss, particularly when attempting 1-ms data intervals.

These issues were mostly related to the synchronization mismatches and the overhead due to DMA re-initialization for each SPI transaction on the Slave.

The introduction of circular DMA on the Slave board along with the framing protocol, CRC-8 calculation, and robust error handling showed significant improvements in the results. Due to these enhancements, the system demonstrated much more stable and reliable communication. Tests involving the transmission of 100,000 or more packets at 1-ms intervals showed successful reception rates around 99%. While occasional packet loss still occurred, the system was generally able to self-recover from transient errors and was able to maintain a continuous data flow for extended periods of time. The RTT, in this context, measured the complete loop from Python sending a UDP command to Python receiving the UDP packet containing the Slave's data. Furthermore, the analysis done with Wireshark indicated that while the immediate SPI turnaround on the microcontrollers was very fast, a significant portion of the observed RTT in the Python application was due to the network stack and OS scheduling delays on the PC running the Python script.

4.4 HIL Phase

After the extensive development and validation of the Ethernet-SPI Bridge using the Python-based prototyping environment (as explained in Section 4.3), the system was shifted to the dSPACE SCALEXIO Hardware-in-the-Loop (HIL) platform. The aim of this phase was to verify the system's functionality, real-time performance, and stability within a HIL simulation environment, utilizing the real-time application and dSPACE ControlDesk interface.

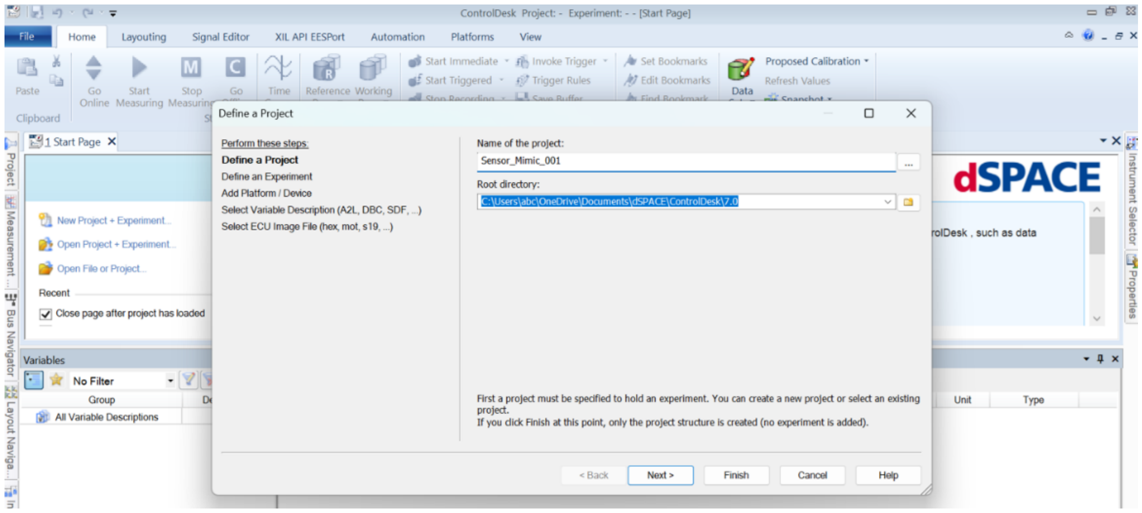
4.4.1 Deployment of the Real-Time Application on SCALEXIO

After the successful generation of the necessary build files (including the .sdf interface definition and .rta real-time application files, as depicted in Figure 3-14: dSPACE ConfigurationDesk Build Results) from the dSPACE ConfigurationDesk environment, the subsequent phase involved deploying this application onto the SCALEXIO HIL platform for testing. This required configuring the personal computer as a host workstation and establishing a reliable network connection with the HIL simulator.

To ensure proper communication, the host workstation's Ethernet network interface was assigned with a static IP address that was 192.168.140.11 and a subnet mask of 255.255.255.0. This configuration was chosen to place the host PC on the same network as the dSPACE SCALEXIO HIL simulator, which had an IP address of 192.168.140.10. This shared network setup is essential for the host workstation to connect to and deploy the real-time application onto the SCALEXIO target. Connectivity between the host PC and the HIL simulator was verified by successfully pinging the simulator's IP address (192.168.140.10) from the Windows command prompt on the host workstation.

With the confirmation of the network connectivity, a new project was created in dSPACE ControlDesk. This project was utilized for loading the real-time application and managing the HIL experiment. The steps followed in the creation of this project are enlisted as:

- A new project was initiated, for example, named "Sensor_Mimic_001" (as shown in the "Name of the project" field in Figure 4-12: dSPACE ControlDesk – Project Creation Steps, top).
- Within this project, a new experiment was defined, which can be the same as the project's name or a distinct identifier.
- The SCALEXIO platform was selected as the target hardware for the experiment (Figure 4-12: dSPACE ControlDesk – Project Creation Steps, middle, shows "SCALEXIO" selected under "Supported Platform/Device Types").
- The crucial step involved linking the experiment to the compiled real-time application by selecting the appropriate Variable Description File (.sdf). As shown in Figure 4-12: dSPACE ControlDesk – Project Creation Steps (bottom), the "Sensor_Mimic_001.sdf" file (generated during the build process detailed in Section 3.3.3) was imported. This file contains all the information about the real-time application's variables, parameters, and signals that can be accessed and manipulated via dSPACE ControlDesk.



Perform these steps:

Define a Project

Define an Experiment

Add Platform / Device

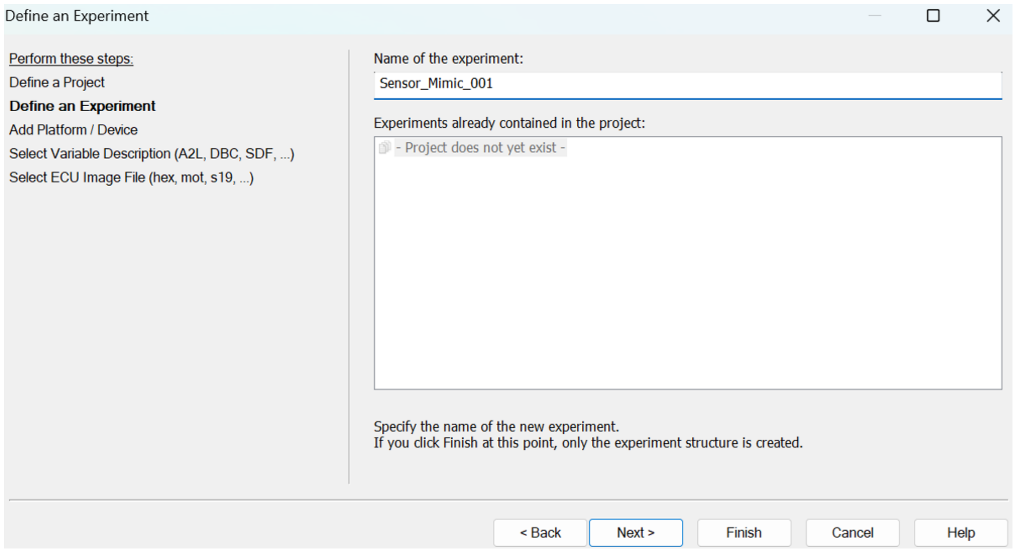
Select Variable Description (A2L, DBC, SDF, ...)

Select ECU Image File (hex, mot, s19, ...)

Name of the project: Sensor_Mimic_001

Root directory: C:\Users\abc\OneDrive\Documents\dSPACE\ControlDesk\7.0

First a project must be specified to hold an experiment. You can create a new project or select an existing project.
If you click Finish at this point, only the project structure is created (no experiment is added).

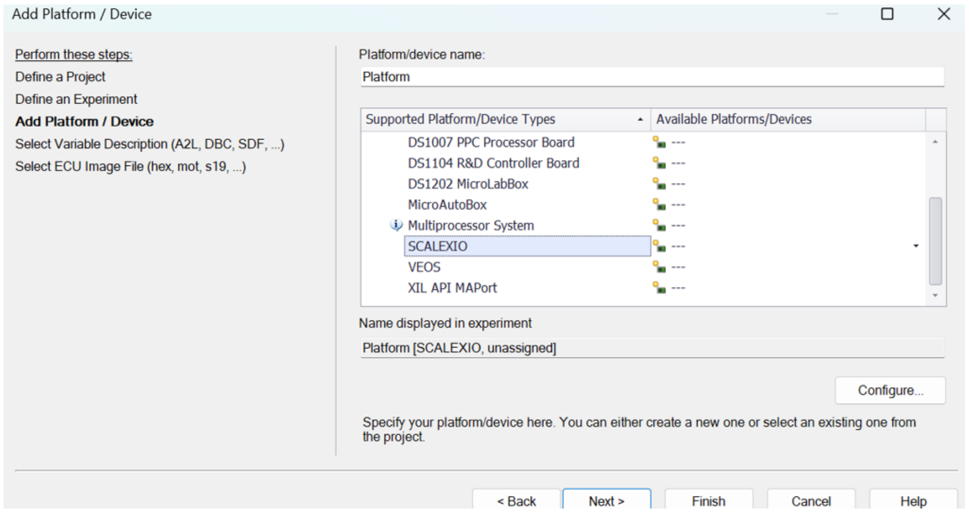

Define an Experiment

Perform these steps:
Define a Project
Define an Experiment
Add Platform / Device
Select Variable Description (A2L, DBC, SDF, ...)
Select ECU Image File (hex, mot, s19, ...)

Name of the experiment:
Sensor_Mimic_001

Experiments already contained in the project:
- Project does not yet exist -

Specify the name of the new experiment.
If you click Finish at this point, only the experiment structure is created.


Add Platform / Device

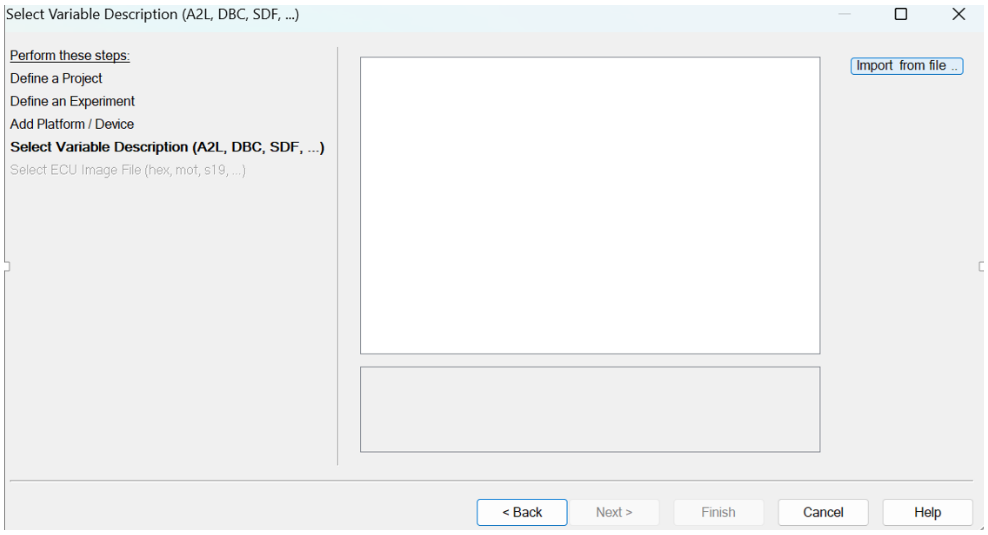
Perform these steps:
Define a Project
Define an Experiment
Add Platform / Device
Select Variable Description (A2L, DBC, SDF, ...)
Select ECU Image File (hex, mot, s19, ...)

Platform/device name:
Platform

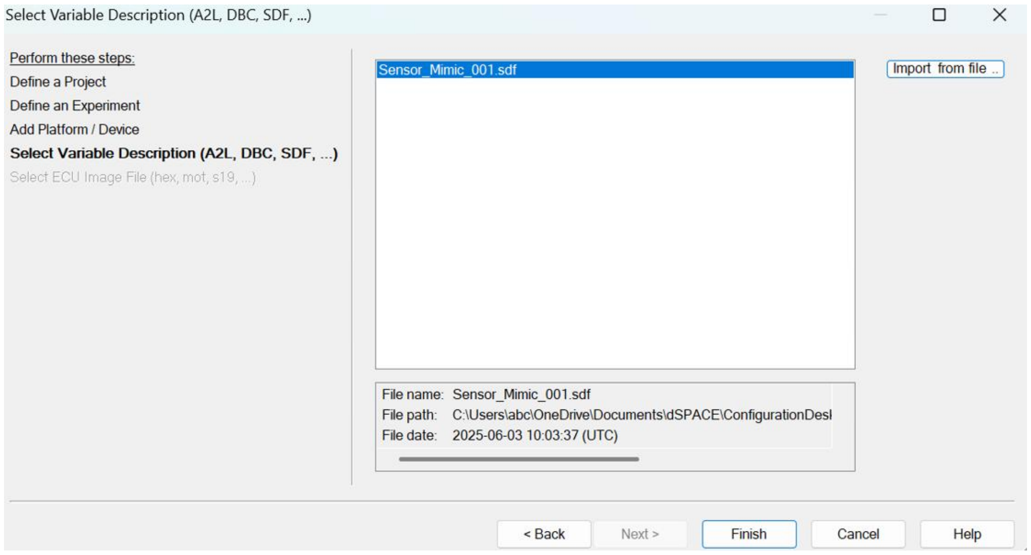
Supported Platform/Device Types	Available Platforms/Devices
DS1007 PPC Processor Board	---
DS1104 R&D Controller Board	---
DS1202 MicroLabBox	---
MicroAutoBox	---
Multiprocessor System	---
SCALEXIO	---
VEOS	---
XIL API MAPort	---

Name displayed in experiment
Platform [SCALEXIO, unassigned]

Specify your platform/device here. You can either create a new one or select an existing one from the project.


Select Variable Description (A2L, DBC, SDF, ...)

Perform these steps:
Define a Project
Define an Experiment
Add Platform / Device
Select Variable Description (A2L, DBC, SDF, ...)
Select ECU Image File (hex, mot, s19, ...)


Select Variable Description (A2L, DBC, SDF, ...)
Perform these steps:
Define a Project
Define an Experiment
Add Platform / Device
Select Variable Description (A2L, DBC, SDF, ...)
Select ECU Image File (hex, mot, s19, ...)
Import from file ..
< Back
Next >
Finish
Cancel
Help

Sensor_Mimic_001.sdf

File name: Sensor_Mimic_001.sdf
File path: C:\Users\abc\OneDrive\Documents\dSPACE\ConfigurationDesl
File date: 2025-06-03 10:03:37 (UTC)

Figure 4-12: dSPACE ControlDesk – Project Creation Steps

Upon successful creation and configuration of the dSPACE ControlDesk project, the real-time application was loaded onto the SCALEXIO platform when going "online." The variables defined within the Simulink model (such as Speed_Input, Speed_Echo, SubCmd_Control, and “y” from MATLAB function *FrameResponse*, as described in Section 3.3.2) became accessible in the "Variables" pane of dSPACE ControlDesk (as depicted in Figure 3-15: dSPACE ControlDesk Layout - Variable and Instrument Selector Pane), ready to be linked to HMI instruments for experiment control and data visualization. This completed the deployment of the real-time application, setting the stage for test execution.

4.4.2 Test Execution Methodology using dSPACE ControlDesk

Once the real-time application was deployed on SCALEXIO, dSPACE ControlDesk was utilized to execute the Simulink models described in section 3.3.2 by interacting with the HMI layouts specifically designed for them.

For the "Speed_Echo_CD" test scenario, the primary objective was to transmit dynamically generated speed values to the STM32 system at 1ms intervals and observe the echoed values returned through the UDP-SPI-UDP bridge. The dSPACE ControlDesk layout featured a slider instrument assigned to the *Speed_Input* variable that is defined as a 'Parameter' of type 'single' in the Simulink model. This allowed the operator to vary the speed input, and the slider range was adjusted to generate these values, typically within a range of 0 to 120 km/h. The numerical value from the slider was processed by the MATLAB Function block (*PackFloatToBytes*) within the real-time application to convert the single-precision float into a 4-element uint8 array before UDP transmission. A display instrument in dSPACE ControlDesk was assigned to the *Speed_Echo* Data Store Write variable (a 'Measurement' of type 'single'), which received its value from the output of the MATLAB Function block (*UnpackBytesToFloat*) after processing the UDP echo from the

STM32 Master. Furthermore, dragging the *Speed_Echo* variable onto the layout automatically generated a time-plot graph, allowing for the visualization of the echoed speed values against time. Activating the "start measurement" function in dSPACE ControlDesk initiated the data flow, allowing for real-time observation of the system's response to varying inputs from the slider, both on the numerical display and the time-plot graph.

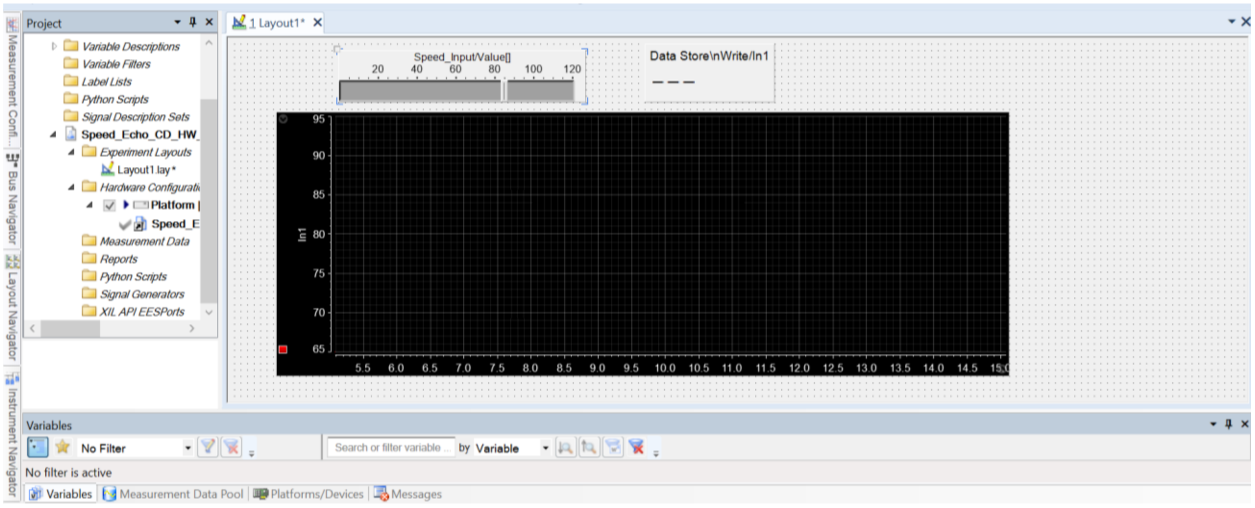


Figure 4-13: dSPACE ControlDesk interface for the Speed Echo test

For the "Speed_Echo_Latency" test scenario, which ran independently inside the Simulink model as described in Section 3.3.2.1, it continuously sends the dynamic speed values and calculates the round-trip time for each successful echo. The objective of this methodology was to capture a long-duration data set of the RTT performance under a sustained 1-ms cycle time. As described in Section 3.3.4, the dSPACE ControlDesk layout was configured to monitor the live *EchoLatency_ms* signal both numerically and graphically, allowing verification of the system's real-time communication characteristics.

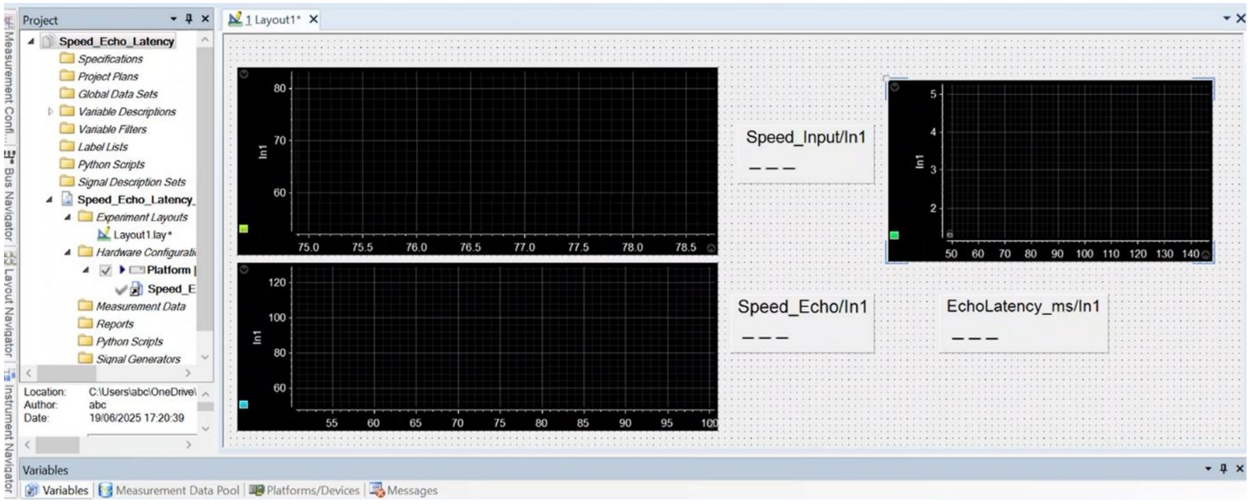
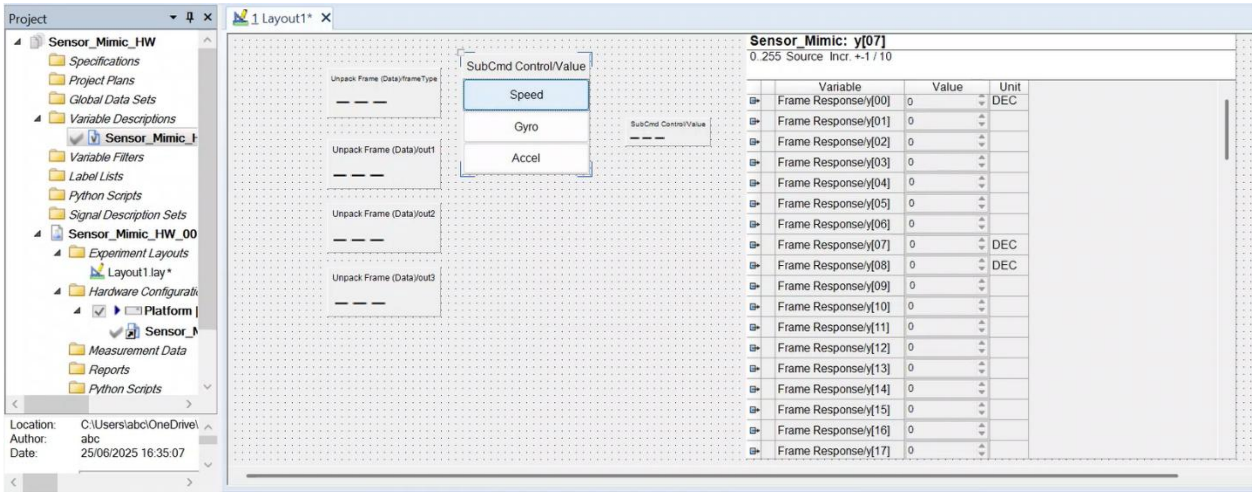


Figure 4-14: dSPACE ControlDesk interface for the Speed Echo Latency test

For the "Sensor_Mimic" test scenario, the dSPACE ControlDesk layout was designed to send 1-byte sub-commands for the reception of the emulated sensor's data from the Slave board. An ON/OFF button was used for this purpose, as its characteristics allow it to send only one command at a time, which aligns with the STM32 system's working operational logic. This instrument allows the operator to send a specific "On" value when pressed and then revert to a default "Off" value. Three buttons were configured for "Stream Speed," "Stream Gyro," and "Stream Accel," with their respective "On" values set to the sub-commands 0x01, 0x02, and 0x03. Moreover, the "Off" value for all buttons was set to *255 (0xFF)*, which the Master STM32 firmware interprets as a generic "poll" command. This setup allowed an operator to send a single mode-change command by clicking a button, after which the HIL would automatically and continuously poll the Slave for data in that newly set mode. Inherently, the ON/OFF-button instrument allows transmission of two signals, either button 1 or button 2. However, this button was configured for transmitting up to three signals (one at a time), corresponding to values 1 through 3, to represent "Stream Speed," "Stream Gyro," and "Stream Accel" modes, respectively. The desired variable, *SubCmd_Control* (variable type "Parameter" and data type uint8), was assigned to the ON/OFF-button. As mentioned in section 3.3.2.3, the responses from the STM32 system were handled by the MATLAB function blocks called *UnpackFrame* and *FormatResponse* in the Simulink model. Numerical displays were linked to the intermediate outputs of this *UnpackFrame* block (frameType, out1, out2, out3) to observe the raw received frame type and unpacked data (hexadecimal or decimal format). This allowed for immediate verification that the Slave first sent a *FRAME_TYPE_SLAVE_ACK* (0xB0) in response to the command and subsequently began streaming the correct data type (0xA1, 0xA2, or 0xA3). The final formatted string from the *FormatResponse* block was displayed in a "Variable Array" instrument for a more human-readable output. This methodology facilitated step-by-step verification of command handling, acknowledgment, and data reception for each emulated sensor mode.



Location: C:\Users\abc\OneDrive\
Author: abc
Date: 25/06/2025 16:35:07

Sensor_Mimic: y[07]
0..255 Source Incr. +-1 / 10

Variable	Value	Unit
Frame Response/y[00]	0	DEC
Frame Response/y[01]	0	
Frame Response/y[02]	0	
Frame Response/y[03]	0	
Frame Response/y[04]	0	
Frame Response/y[05]	0	
Frame Response/y[06]	0	
Frame Response/y[07]	0	DEC
Frame Response/y[08]	0	DEC
Frame Response/y[09]	0	
Frame Response/y[10]	0	
Frame Response/y[11]	0	
Frame Response/y[12]	0	
Frame Response/y[13]	0	
Frame Response/y[14]	0	
Frame Response/y[15]	0	
Frame Response/y[16]	0	
Frame Response/y[17]	0	

Figure 4-15: dSPACE ControlDesk interface for the Sensor Mimic test

The successful execution of these tests by utilizing dSPACE ControlDesk provided a reliable method for validating the proper functionality and real-time behavior of the Ethernet-SPI communication bridge.

Chapter 5

Testing and Results

5.1 Prototype Testing: Python-Driven Ethernet-SPI Bridge

The initial and thorough testing phase used a Python application to mimic the HIL simulator and carefully check the end-to-end functionality of the Ethernet-SPI Bridge. This approach provided precise control over the test conditions and allowed rapid iteration with detailed logging. The Master STM32 board received UDP commands from the Python script, processed them, and then sent them over to the Slave STM32 board via the custom SPI protocol and then relayed the Slave's responses back to the Python application via UDP. This setup facilitated the analysis of command handling, data integrity, SPI frame parsing, CRC validation, sequence management, error recovery, and communication latency (round-trip time—RTT).

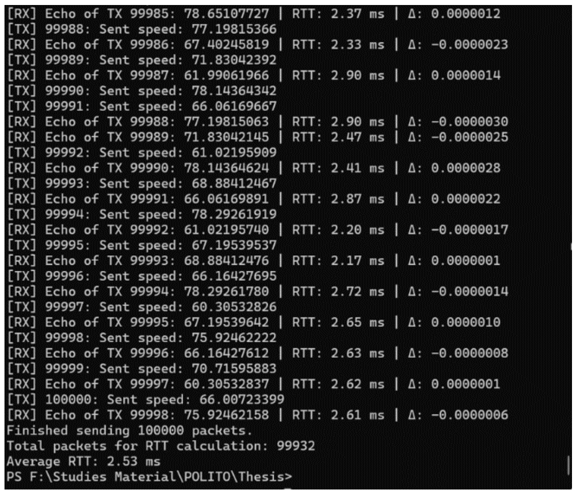
5.1.1 Python Application Analysis

The Python application, equipped with a Tkinter GUI for interactive command sending and displaying the output log, is used as a main tool for performing the functional testing of the prototype.

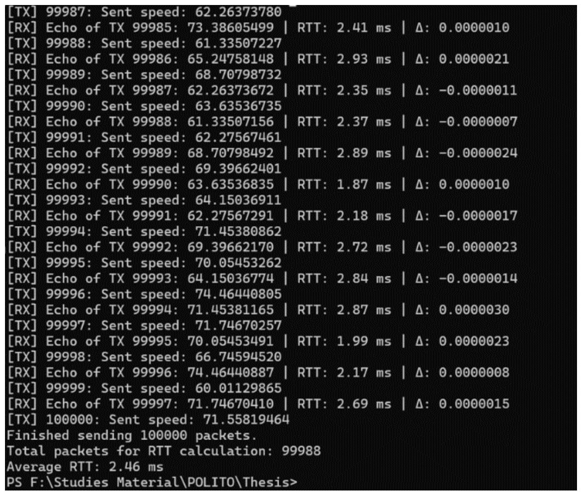
5.1.1.1 Basic UDP-SPI Echo and Latency Testing

The foundational tests focused on verifying the simplest end-to-end data communication path, where a Python script is used to send a floating-point "speed" value via UDP. The Master STM32 receives this speed value and transmits it to the Slave STM32 via SPI, then the Slave echoes this value back to the Master via SPI, and finally the Master sends this echoed speed value back to the Python application via UDP. These tests were conducted at sending intervals of 1-ms and subsequently 0.5-ms from the Python script to evaluate the system's responsiveness and basic data handling capacity. The Python application logged the sent speed, the received echoed speed, and calculated the RTT for each transaction.

The system was tested for a 1-ms sending interval, and it demonstrated successful echo of speed values. RTTs were logged that provided initial benchmarks for the communication loop. Packet loss was observed but was much less than 0.1 percent. The average RTT value for these transactions done at 1-ms was around 2.5-ms. Results of these tests conducted at a sending rate of 1-ms are hereby depicted in Figure 5-1: Prototype Basic UDP Communication Results at 1-ms (Python Console).



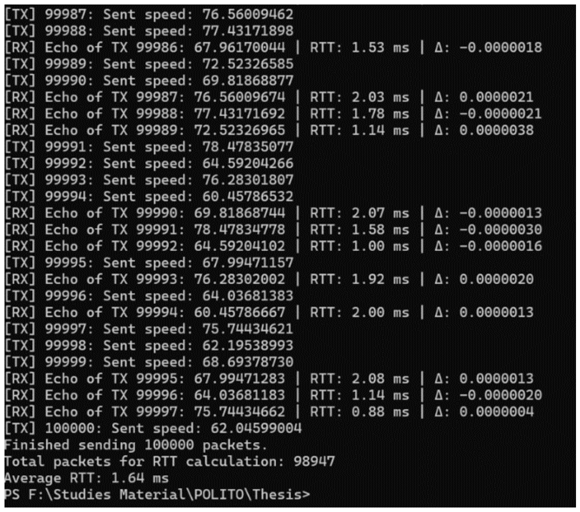
```
[RX] Echo of TX 99985: 78.65107727 | RTT: 2.37 ms | Δ: 0.0000012
[TX] 99988: Sent speed: 77.19815366
[RX] Echo of TX 99986: 67.40245819 | RTT: 2.33 ms | Δ: -0.0000023
[TX] 99989: Sent speed: 71.83042392
[RX] Echo of TX 99987: 61.99061966 | RTT: 2.90 ms | Δ: 0.0000014
[TX] 99990: Sent speed: 78.14364342
[TX] 99991: Sent speed: 66.06169667
[RX] Echo of TX 99988: 77.19815063 | RTT: 2.90 ms | Δ: -0.0000030
[RX] Echo of TX 99989: 71.83042145 | RTT: 2.47 ms | Δ: -0.0000025
[TX] 99992: Sent speed: 61.02195909
[RX] Echo of TX 99990: 78.14364624 | RTT: 2.41 ms | Δ: 0.0000028
[TX] 99993: Sent speed: 68.88412467
[RX] Echo of TX 99991: 66.06169891 | RTT: 2.87 ms | Δ: 0.0000022
[TX] 99994: Sent speed: 78.29261919
[RX] Echo of TX 99992: 61.02195740 | RTT: 2.20 ms | Δ: -0.0000017
[TX] 99995: Sent speed: 67.19539537
[RX] Echo of TX 99993: 68.88412476 | RTT: 2.17 ms | Δ: 0.0000001
[TX] 99996: Sent speed: 66.16427695
[RX] Echo of TX 99994: 78.29261780 | RTT: 2.72 ms | Δ: -0.0000014
[TX] 99997: Sent speed: 60.30532826
[RX] Echo of TX 99995: 67.19539642 | RTT: 2.65 ms | Δ: 0.0000010
[TX] 99998: Sent speed: 75.92462222
[RX] Echo of TX 99996: 66.16427612 | RTT: 2.63 ms | Δ: -0.0000008
[TX] 99999: Sent speed: 70.71595883
[RX] Echo of TX 99997: 60.30532837 | RTT: 2.62 ms | Δ: 0.0000001
[TX] 100000: Sent speed: 66.00723399
[RX] Echo of TX 99998: 75.92462158 | RTT: 2.61 ms | Δ: -0.0000006
Finished sending 100000 packets.
Total packets for RTT calculation: 99932
Average RTT: 2.53 ms
PS F:\Studies Material\POLITO\Thesis>
```



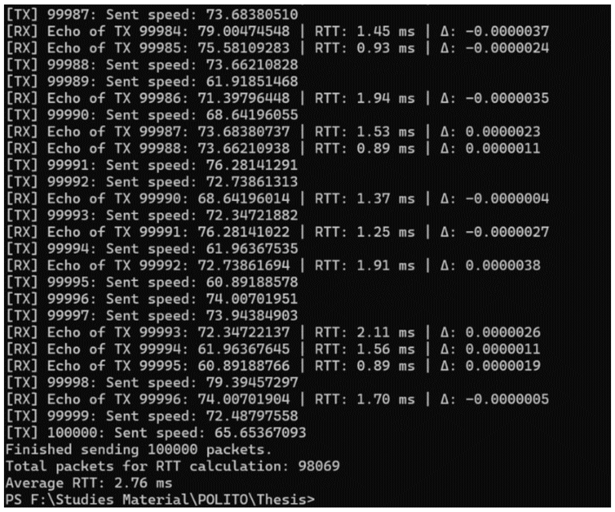
```
[TX] 99987: Sent speed: 62.26373780
[RX] Echo of TX 99985: 73.38605499 | RTT: 2.41 ms | Δ: 0.0000010
[TX] 99988: Sent speed: 61.33507227
[RX] Echo of TX 99986: 65.24758148 | RTT: 2.93 ms | Δ: 0.0000021
[TX] 99989: Sent speed: 68.70798732
[RX] Echo of TX 99987: 62.26373672 | RTT: 2.35 ms | Δ: -0.0000011
[TX] 99990: Sent speed: 63.63536735
[RX] Echo of TX 99988: 61.33507156 | RTT: 2.37 ms | Δ: -0.0000007
[TX] 99991: Sent speed: 62.27567461
[RX] Echo of TX 99989: 68.70798492 | RTT: 2.89 ms | Δ: -0.0000024
[TX] 99992: Sent speed: 69.39662401
[RX] Echo of TX 99990: 63.63536835 | RTT: 1.87 ms | Δ: 0.0000010
[TX] 99993: Sent speed: 64.15036911
[RX] Echo of TX 99991: 62.27567291 | RTT: 2.18 ms | Δ: -0.0000017
[TX] 99994: Sent speed: 71.45380862
[RX] Echo of TX 99992: 69.39662170 | RTT: 2.72 ms | Δ: -0.0000023
[TX] 99995: Sent speed: 70.05453262
[RX] Echo of TX 99993: 64.15036774 | RTT: 2.84 ms | Δ: -0.0000014
[TX] 99996: Sent speed: 74.46440805
[RX] Echo of TX 99994: 71.45381165 | RTT: 2.87 ms | Δ: 0.0000030
[TX] 99997: Sent speed: 71.74670257
[RX] Echo of TX 99995: 70.05453491 | RTT: 1.99 ms | Δ: 0.0000023
[TX] 99998: Sent speed: 66.74594520
[RX] Echo of TX 99996: 74.46440887 | RTT: 2.17 ms | Δ: 0.0000008
[TX] 99999: Sent speed: 60.01129865
[RX] Echo of TX 99997: 71.74670410 | RTT: 2.69 ms | Δ: 0.0000015
[TX] 100000: Sent speed: 71.55819464
Finished sending 100000 packets.
Total packets for RTT calculation: 99988
Average RTT: 2.46 ms
PS F:\Studies Material\POLITO\Thesis>
```

Figure 5-1: Prototype Basic UDP Communication Results at 1-ms (Python Console)

The same setup was also tested for a 0.5-ms sending interval, and it demonstrated successful echo of speed values. RTTs were logged that provided initial benchmarks for the communication loop. More packet loss was observed but was less than 2 percent. The average RTT value for these transactions done at 0.5-ms was around 2-ms. Results of these tests conducted at a sending rate of 0.5-ms are hereby depicted in Figure 5-2: Prototype Basic UDP Communication Results at 0.5-ms (Python Console). Overall, pushing the interval to 0.5-ms stressed the system, and the packet loss increased.



```
[TX] 99987: Sent speed: 76.56009462
[TX] 99988: Sent speed: 77.43171898
[RX] Echo of TX 99986: 67.96170044 | RTT: 1.53 ms | Δ: -0.0000018
[TX] 99989: Sent speed: 72.52326585
[TX] 99990: Sent speed: 69.81868877
[RX] Echo of TX 99987: 76.56009674 | RTT: 2.03 ms | Δ: 0.0000021
[RX] Echo of TX 99988: 77.43171692 | RTT: 1.78 ms | Δ: -0.0000021
[RX] Echo of TX 99989: 72.52326965 | RTT: 1.14 ms | Δ: 0.0000038
[TX] 99991: Sent speed: 78.47835077
[TX] 99992: Sent speed: 64.59204266
[TX] 99993: Sent speed: 76.28301807
[TX] 99994: Sent speed: 60.45786532
[RX] Echo of TX 99990: 69.81868744 | RTT: 2.07 ms | Δ: -0.0000013
[RX] Echo of TX 99991: 78.47834778 | RTT: 1.58 ms | Δ: -0.0000030
[RX] Echo of TX 99992: 64.59204102 | RTT: 1.00 ms | Δ: -0.0000016
[TX] 99995: Sent speed: 67.99471157
[RX] Echo of TX 99993: 76.28302002 | RTT: 1.92 ms | Δ: 0.0000020
[TX] 99996: Sent speed: 64.03681383
[RX] Echo of TX 99994: 60.45786667 | RTT: 2.00 ms | Δ: 0.0000013
[TX] 99997: Sent speed: 75.74434621
[TX] 99998: Sent speed: 62.19538993
[TX] 99999: Sent speed: 68.69378730
[RX] Echo of TX 99995: 67.99471283 | RTT: 2.08 ms | Δ: 0.0000013
[RX] Echo of TX 99996: 64.03681183 | RTT: 1.14 ms | Δ: -0.0000020
[RX] Echo of TX 99997: 75.74434662 | RTT: 0.88 ms | Δ: 0.0000004
[TX] 100000: Sent speed: 62.04599004
Finished sending 100000 packets.
Total packets for RTT calculation: 98947
Average RTT: 1.64 ms
PS F:\Studies Material\POLITO\Thesis>
```



```
[TX] 99987: Sent speed: 73.68380510
[RX] Echo of TX 99984: 79.00474548 | RTT: 1.45 ms | Δ: -0.0000037
[RX] Echo of TX 99985: 75.58109283 | RTT: 0.93 ms | Δ: -0.0000024
[TX] 99988: Sent speed: 73.66210828
[TX] 99989: Sent speed: 61.91851468
[RX] Echo of TX 99986: 71.39796448 | RTT: 1.94 ms | Δ: -0.0000035
[TX] 99990: Sent speed: 68.64196055
[RX] Echo of TX 99987: 73.68380737 | RTT: 1.53 ms | Δ: 0.0000023
[RX] Echo of TX 99988: 73.66210938 | RTT: 0.89 ms | Δ: 0.0000011
[TX] 99991: Sent speed: 76.28141291
[TX] 99992: Sent speed: 72.73861313
[RX] Echo of TX 99990: 68.64196014 | RTT: 1.37 ms | Δ: -0.0000004
[TX] 99993: Sent speed: 72.34721882
[RX] Echo of TX 99991: 76.28141022 | RTT: 1.25 ms | Δ: -0.0000027
[TX] 99994: Sent speed: 61.96367535
[RX] Echo of TX 99992: 72.73861694 | RTT: 1.91 ms | Δ: 0.0000038
[TX] 99995: Sent speed: 60.89188578
[TX] 99996: Sent speed: 74.00701951
[TX] 99997: Sent speed: 73.94384903
[RX] Echo of TX 99993: 72.34722137 | RTT: 2.11 ms | Δ: 0.0000026
[RX] Echo of TX 99994: 61.96367645 | RTT: 1.56 ms | Δ: 0.0000011
[RX] Echo of TX 99995: 60.89188766 | RTT: 0.89 ms | Δ: 0.0000019
[TX] 99998: Sent speed: 79.39457297
[RX] Echo of TX 99996: 74.00701904 | RTT: 1.70 ms | Δ: -0.0000005
[TX] 99999: Sent speed: 72.48797558
[TX] 100000: Sent speed: 65.65367093
Finished sending 100000 packets.
Total packets for RTT calculation: 98069
Average RTT: 2.76 ms
PS F:\Studies Material\POLITO\Thesis>
```

Figure 5-2: Prototype Basic UDP Communication Results at 0.5-ms (Python Console)

Table 5-1: Prototype Basic UDP Communication Results summarizes the results achieved for the tests conducted for basic UDP communication.

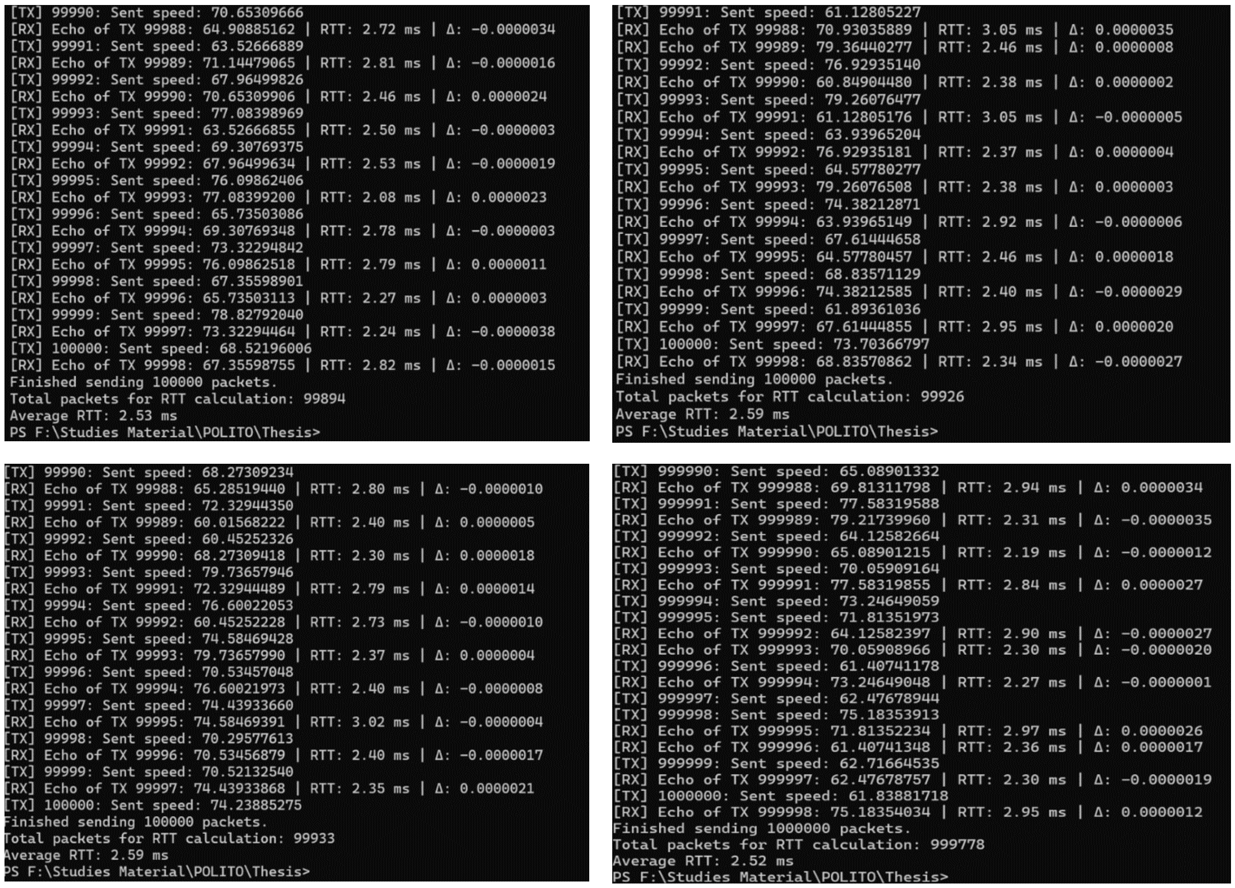
Table 5-1: Prototype Basic UDP Communication Results

Sending Rate	Packets sent	Matched Echoes	Packet Loss (%)	Avg RTT (ms)
1-ms	100000	99932	0.068	2.53
	100000	99988	0.012	2.46
0.5-ms	100000	98947	1.053	1.64
	100000	98069	1.931	2.76

5.1.1.2 Validating Custom SPI Protocol with Framing and CRC-8

After the initial echo tests, the custom SPI protocol, incorporating framing (*START_BYTE, LENGTH, TYPE, END_BYTE*) and software-implemented CRC-8, was integrated into both Master and Slave firmware. The same Python application was used to send floating-point speed values. The Master now framed this speed data with the customized protocol (e.g., *FRAME_TYPE_MASTER_DATA*) before SPI transmission to the Slave. The Slave, upon receiving and validating the framed data (including CRC check), would frame its echo response (e.g., *FRAME_TYPE_SLAVE_ECHO*) similarly. The Master then parsed the Slave's framed response, validated its CRC, and forwarded the payload to Python via UDP. Tests were again conducted at 1-ms and 0.5-ms sending intervals.

Tests conducted at 1-ms sending interval showed almost the similar results like the basic UDP communication tests. The average RTT and the percentage of packets lost lied in the same range like the tests conducted without the implementation of customized SPI protocol and software-based CRC-8. Figure 5-3: Prototype Echo Test (Custom SPI + CRC-8) Results at 1-ms (Python Console) shows the results conducted of the tests conducted at 1-ms.



```
[TX] 99990: Sent speed: 70.65309666
[RX] Echo of TX 99988: 64.90885162 | RTT: 2.72 ms | Δ: -0.0000034
[TX] 99991: Sent speed: 63.52666889
[RX] Echo of TX 99989: 71.14479065 | RTT: 2.81 ms | Δ: -0.0000016
[TX] 99992: Sent speed: 67.96499826
[RX] Echo of TX 99990: 70.65309906 | RTT: 2.46 ms | Δ: 0.0000024
[TX] 99993: Sent speed: 77.08398969
[RX] Echo of TX 99991: 63.52666855 | RTT: 2.50 ms | Δ: -0.0000003
[TX] 99994: Sent speed: 69.30769375
[RX] Echo of TX 99992: 67.96499634 | RTT: 2.53 ms | Δ: -0.0000019
[TX] 99995: Sent speed: 76.09862406
[RX] Echo of TX 99993: 77.08399200 | RTT: 2.08 ms | Δ: 0.0000023
[TX] 99996: Sent speed: 65.73503086
[RX] Echo of TX 99994: 69.30769348 | RTT: 2.78 ms | Δ: -0.0000003
[TX] 99997: Sent speed: 73.32294842
[RX] Echo of TX 99995: 76.09862518 | RTT: 2.79 ms | Δ: 0.0000011
[TX] 99998: Sent speed: 67.35598901
[RX] Echo of TX 99996: 65.73503113 | RTT: 2.27 ms | Δ: 0.0000003
[TX] 99999: Sent speed: 78.82792040
[RX] Echo of TX 99997: 73.32294464 | RTT: 2.24 ms | Δ: -0.0000038
[TX] 100000: Sent speed: 68.52196006
[RX] Echo of TX 99998: 67.35598755 | RTT: 2.82 ms | Δ: -0.0000015
Finished sending 100000 packets.
Total packets for RTT calculation: 99894
Average RTT: 2.53 ms
PS F:\Studies Material\POLITO\Thesis>
```

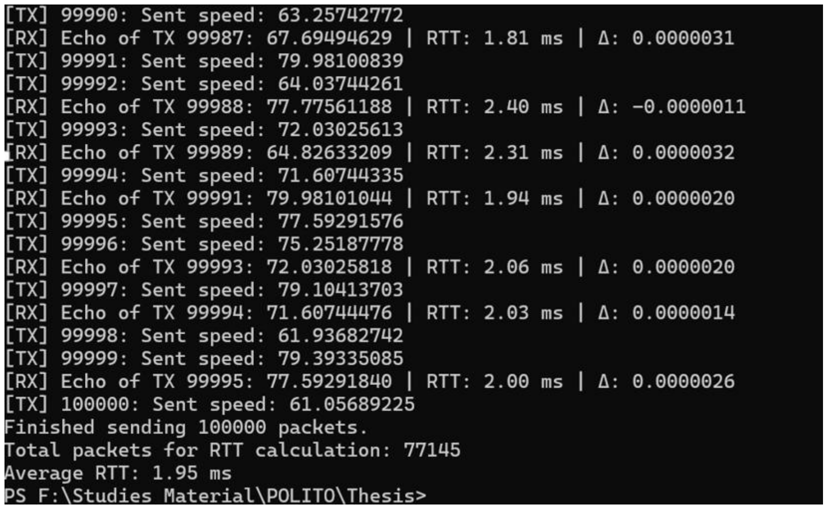
```
[TX] 99991: Sent speed: 61.12805227
[RX] Echo of TX 99988: 70.93035889 | RTT: 3.05 ms | Δ: 0.0000035
[RX] Echo of TX 99989: 79.36440277 | RTT: 2.46 ms | Δ: 0.0000008
[TX] 99992: Sent speed: 76.92935140
[RX] Echo of TX 99990: 60.84904480 | RTT: 2.38 ms | Δ: 0.0000002
[TX] 99993: Sent speed: 79.26076477
[RX] Echo of TX 99991: 61.12805176 | RTT: 3.05 ms | Δ: -0.0000005
[TX] 99994: Sent speed: 63.93965204
[RX] Echo of TX 99992: 76.92935181 | RTT: 2.37 ms | Δ: 0.0000004
[TX] 99995: Sent speed: 64.57780277
[RX] Echo of TX 99993: 79.26076508 | RTT: 2.38 ms | Δ: 0.0000003
[TX] 99996: Sent speed: 74.38212871
[RX] Echo of TX 99994: 63.93965149 | RTT: 2.92 ms | Δ: -0.0000006
[TX] 99997: Sent speed: 67.61444658
[RX] Echo of TX 99995: 64.57780457 | RTT: 2.46 ms | Δ: 0.0000018
[TX] 99998: Sent speed: 68.83571129
[RX] Echo of TX 99996: 74.38212585 | RTT: 2.40 ms | Δ: -0.0000029
[TX] 99999: Sent speed: 61.89361036
[RX] Echo of TX 99997: 67.61444855 | RTT: 2.95 ms | Δ: 0.0000020
[TX] 100000: Sent speed: 73.70366797
[RX] Echo of TX 99998: 68.83570862 | RTT: 2.34 ms | Δ: -0.0000027
Finished sending 100000 packets.
Total packets for RTT calculation: 99926
Average RTT: 2.59 ms
PS F:\Studies Material\POLITO\Thesis>
```

```
[TX] 99990: Sent speed: 68.27309234
[RX] Echo of TX 99988: 65.28519440 | RTT: 2.80 ms | Δ: -0.0000010
[TX] 99991: Sent speed: 72.32944350
[RX] Echo of TX 99989: 60.01568222 | RTT: 2.40 ms | Δ: 0.0000005
[TX] 99992: Sent speed: 60.45252326
[RX] Echo of TX 99990: 68.27309418 | RTT: 2.30 ms | Δ: 0.0000018
[TX] 99993: Sent speed: 79.73657946
[RX] Echo of TX 99991: 72.32944489 | RTT: 2.79 ms | Δ: 0.0000014
[TX] 99994: Sent speed: 76.60022053
[RX] Echo of TX 99992: 60.45252228 | RTT: 2.73 ms | Δ: -0.0000010
[TX] 99995: Sent speed: 74.58469428
[RX] Echo of TX 99993: 79.73657990 | RTT: 2.37 ms | Δ: 0.0000004
[TX] 99996: Sent speed: 70.53457048
[RX] Echo of TX 99994: 76.60021973 | RTT: 2.40 ms | Δ: -0.0000008
[TX] 99997: Sent speed: 74.43933660
[RX] Echo of TX 99995: 74.58469391 | RTT: 3.02 ms | Δ: -0.0000004
[TX] 99998: Sent speed: 70.29577613
[RX] Echo of TX 99996: 70.53456879 | RTT: 2.40 ms | Δ: -0.0000017
[TX] 99999: Sent speed: 70.52132540
[RX] Echo of TX 99997: 74.43933868 | RTT: 2.35 ms | Δ: 0.0000021
[TX] 100000: Sent speed: 74.23885275
Finished sending 100000 packets.
Total packets for RTT calculation: 99933
Average RTT: 2.59 ms
PS F:\Studies Material\POLITO\Thesis>
```

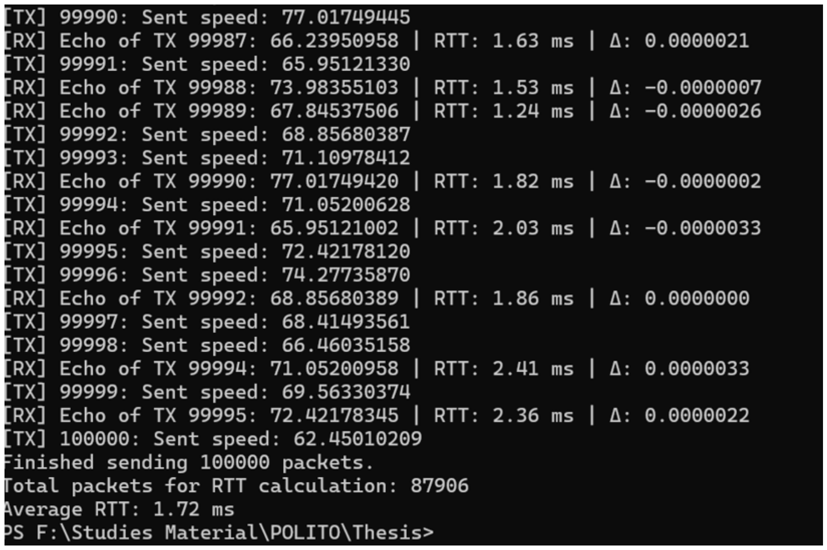
```
[TX] 999990: Sent speed: 65.08901332
[RX] Echo of TX 999988: 69.81311798 | RTT: 2.94 ms | Δ: 0.0000034
[TX] 999991: Sent speed: 77.58319588
[RX] Echo of TX 999989: 79.21739960 | RTT: 2.31 ms | Δ: -0.0000035
[TX] 999992: Sent speed: 64.12582664
[RX] Echo of TX 999990: 65.08901215 | RTT: 2.19 ms | Δ: -0.0000012
[TX] 999993: Sent speed: 70.05909164
[RX] Echo of TX 999991: 77.58319855 | RTT: 2.84 ms | Δ: 0.0000027
[TX] 999994: Sent speed: 73.24649059
[TX] 999995: Sent speed: 71.81351973
[RX] Echo of TX 999992: 64.12582397 | RTT: 2.90 ms | Δ: -0.0000027
[RX] Echo of TX 999993: 70.05908966 | RTT: 2.30 ms | Δ: -0.0000020
[TX] 999996: Sent speed: 61.40741178
[RX] Echo of TX 999994: 73.24649048 | RTT: 2.27 ms | Δ: -0.0000001
[TX] 999997: Sent speed: 62.47678944
[TX] 999998: Sent speed: 75.18353913
[RX] Echo of TX 999995: 71.81352234 | RTT: 2.97 ms | Δ: 0.0000026
[RX] Echo of TX 999996: 61.40741348 | RTT: 2.36 ms | Δ: 0.0000017
[TX] 999999: Sent speed: 62.71664535
[RX] Echo of TX 999997: 62.47678757 | RTT: 2.30 ms | Δ: -0.0000019
[TX] 1000000: Sent speed: 61.83881718
[RX] Echo of TX 999998: 75.18354034 | RTT: 2.95 ms | Δ: 0.0000012
Finished sending 1000000 packets.
Total packets for RTT calculation: 999778
Average RTT: 2.52 ms
PS F:\Studies Material\POLITO\Thesis>
```

Figure 5-3: Prototype Echo Test (Custom SPI + CRC-8) Results at 1-ms (Python Console)

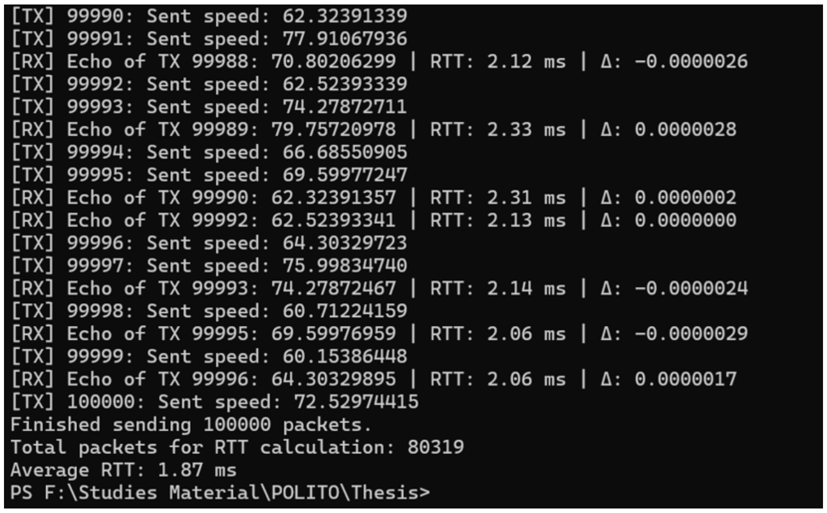
Tests conducted for the custom SPI with CRC system at a sending interval of 0.5-ms showed significant packet loss compared to the test conducted for the basic UDP communication. The average RTT lay in the range of 1.7 to 1.9-ms, but the packet loss was more than 10 percent. Figure 5-4: Prototype Echo Test (Custom SPI + CRC-8) Results at 0.5-ms (Python Console) shows the results conducted of the tests conducted at 0.5-ms.



```
[TX] 99990: Sent speed: 63.25742772
[RX] Echo of TX 99987: 67.69494629 | RTT: 1.81 ms | Δ: 0.0000031
[TX] 99991: Sent speed: 79.98100839
[TX] 99992: Sent speed: 64.03744261
[RX] Echo of TX 99988: 77.77561188 | RTT: 2.40 ms | Δ: -0.0000011
[TX] 99993: Sent speed: 72.03025613
[RX] Echo of TX 99989: 64.82633209 | RTT: 2.31 ms | Δ: 0.0000032
[TX] 99994: Sent speed: 71.60744335
[RX] Echo of TX 99991: 79.98101044 | RTT: 1.94 ms | Δ: 0.0000020
[TX] 99995: Sent speed: 77.59291576
[TX] 99996: Sent speed: 75.25187778
[RX] Echo of TX 99993: 72.03025818 | RTT: 2.06 ms | Δ: 0.0000020
[TX] 99997: Sent speed: 79.10413703
[RX] Echo of TX 99994: 71.60744476 | RTT: 2.03 ms | Δ: 0.0000014
[TX] 99998: Sent speed: 61.93682742
[TX] 99999: Sent speed: 79.39335085
[RX] Echo of TX 99995: 77.59291840 | RTT: 2.00 ms | Δ: 0.0000026
[TX] 100000: Sent speed: 61.05689225
Finished sending 100000 packets.
Total packets for RTT calculation: 77145
Average RTT: 1.95 ms
PS F:\Studies Material\POLITO\Thesis>
```



```
[TX] 99990: Sent speed: 77.01749445
[RX] Echo of TX 99987: 66.23950958 | RTT: 1.63 ms | Δ: 0.0000021
[TX] 99991: Sent speed: 65.95121330
[RX] Echo of TX 99988: 73.98355103 | RTT: 1.53 ms | Δ: -0.0000007
[RX] Echo of TX 99989: 67.84537506 | RTT: 1.24 ms | Δ: -0.0000026
[TX] 99992: Sent speed: 68.85680387
[TX] 99993: Sent speed: 71.10978412
[RX] Echo of TX 99990: 77.01749420 | RTT: 1.82 ms | Δ: -0.0000002
[TX] 99994: Sent speed: 71.05200628
[RX] Echo of TX 99991: 65.95121002 | RTT: 2.03 ms | Δ: -0.0000033
[TX] 99995: Sent speed: 72.42178120
[TX] 99996: Sent speed: 74.27735870
[RX] Echo of TX 99992: 68.85680389 | RTT: 1.86 ms | Δ: 0.0000000
[TX] 99997: Sent speed: 68.41493561
[TX] 99998: Sent speed: 66.46035158
[RX] Echo of TX 99994: 71.05200958 | RTT: 2.41 ms | Δ: 0.0000033
[TX] 99999: Sent speed: 69.56330374
[RX] Echo of TX 99995: 72.42178345 | RTT: 2.36 ms | Δ: 0.0000022
[TX] 100000: Sent speed: 62.45010209
Finished sending 100000 packets.
Total packets for RTT calculation: 87906
Average RTT: 1.72 ms
PS F:\Studies Material\POLITO\Thesis>
```



```
[TX] 99990: Sent speed: 62.32391339
[TX] 99991: Sent speed: 77.91067936
[RX] Echo of TX 99988: 70.80206299 | RTT: 2.12 ms | Δ: -0.0000026
[TX] 99992: Sent speed: 62.52393339
[TX] 99993: Sent speed: 74.27872711
[RX] Echo of TX 99989: 79.75720978 | RTT: 2.33 ms | Δ: 0.0000028
[TX] 99994: Sent speed: 66.68550905
[TX] 99995: Sent speed: 69.59977247
[RX] Echo of TX 99990: 62.32391357 | RTT: 2.31 ms | Δ: 0.0000002
[RX] Echo of TX 99992: 62.52393341 | RTT: 2.13 ms | Δ: 0.0000000
[TX] 99996: Sent speed: 64.30329723
[TX] 99997: Sent speed: 75.99834740
[RX] Echo of TX 99993: 74.27872467 | RTT: 2.14 ms | Δ: -0.0000024
[TX] 99998: Sent speed: 60.71224159
[RX] Echo of TX 99995: 69.59976959 | RTT: 2.06 ms | Δ: -0.0000029
[TX] 99999: Sent speed: 60.15386448
[RX] Echo of TX 99996: 64.30329895 | RTT: 2.06 ms | Δ: 0.0000017
[TX] 100000: Sent speed: 72.52974415
Finished sending 100000 packets.
Total packets for RTT calculation: 80319
Average RTT: 1.87 ms
PS F:\Studies Material\POLITO\Thesis>
```

Figure 5-4: Prototype Echo Test (Custom SPI + CRC-8) Results at 0.5-ms (Python Console)

Table 5-2: Prototype Echo Test (Custom SPI + CRC-8) Results summarizes the results achieved for the tests conducted on the prototype setup with custom SPI and CRC-8 implemented. From the results we can deduce that the overheads added by framing and CRC-8 calculations led to the significant loss in packets in the case of tests conducted at a 0.5-ms sending interval.

Table 5-2: Prototype Echo Test (Custom SPI + CRC-8) Results

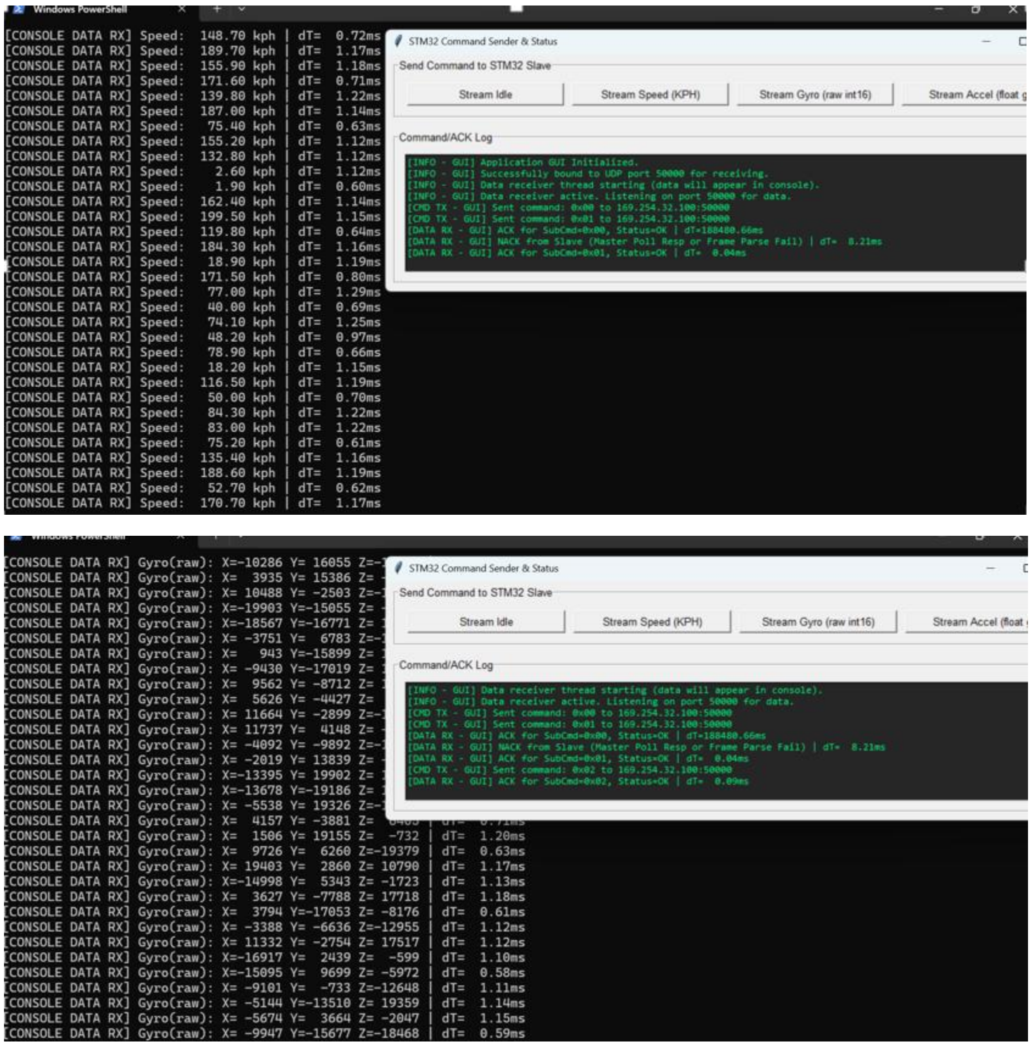
Sending Rate	Packets sent	Matched Echoes	Packet Loss (%)	Avg RTT (ms)
1-ms	100000	99894	0.106	2.53
	100000	99926	0.074	2.59
	100000	99933	0.067	2.59
	1000000	999778	0.022	2.52
0.5-ms	100000	77145	22.86	1.95
	100000	87906	12.09	1.72
	100000	80319	19.68	1.87

5.1.1.3 Commanded Sensor Data Streaming and ACK/NACK Handling

The most comprehensive prototype tests involved the Python GUI sending 1-byte sub-commands to the Master to control the type of data stream emulated by the Slave (Idle, Speed, Gyro, Accel). The Master relayed these commands via SPI using *FRAME_TYPE_CMD_CONTROL_SLAVE*. The Slave responded with either *FRAME_TYPE_SLAVE_ACK* / *FRAME_TYPE_SLAVE_NACK* or, if a data stream was active and the Master sent a poll (0xFF sub-command), with the corresponding sensor data frame (e.g., *FRAME_TYPE_DATA_VEHICLE_SPEED*). The Master forwarded these structured responses (prefixed with the Slave's frame type) to the Python GUI. The Python GUI successfully demonstrated the ability to switch between different sensor data streams:

- The GUI's log area displayed ACK/NACK responses from the Slave, confirming command reception and processing.
- Continuously updating Speed (float), Gyroscope (3x int16_t), and Accelerometer (3x float) data when those respective streams were active.
- Time taken per packet received was demonstrated on the Python console for all the emulated sensors data.

Figure 5-5: UDP Command-Response Prototype Testing Results shows command response results of the prototype system receiving data at an interval of 1-ms per transaction.

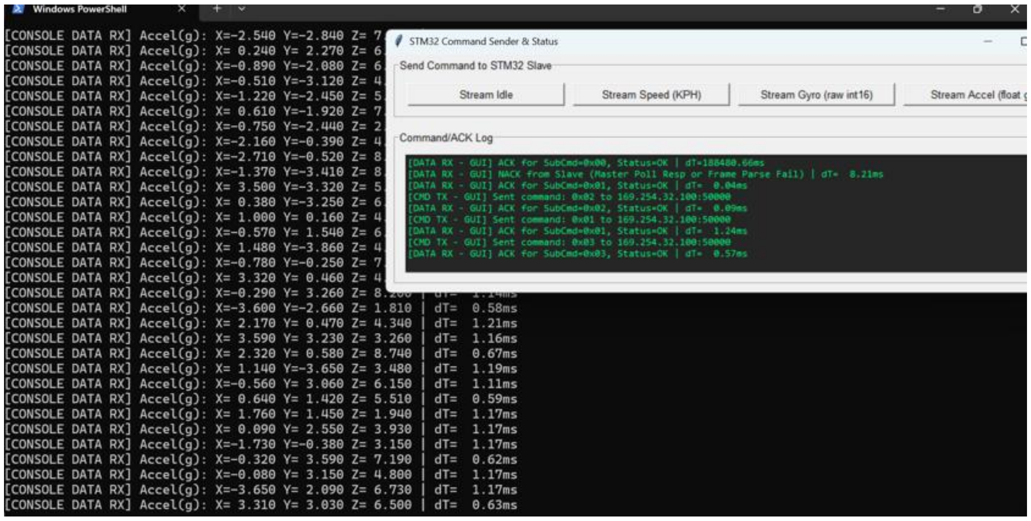


```
[CONSOLE DATA RX] Speed:  148.70 kph | dT=  0.72ms
[CONSOLE DATA RX] Speed:  189.70 kph | dT=  1.17ms
[CONSOLE DATA RX] Speed:  155.90 kph | dT=  1.18ms
[CONSOLE DATA RX] Speed:  171.60 kph | dT=  0.71ms
[CONSOLE DATA RX] Speed:  139.80 kph | dT=  1.22ms
[CONSOLE DATA RX] Speed:  187.00 kph | dT=  1.14ms
[CONSOLE DATA RX] Speed:   75.40 kph | dT=  0.63ms
[CONSOLE DATA RX] Speed:  155.20 kph | dT=  1.12ms
[CONSOLE DATA RX] Speed:  132.80 kph | dT=  1.12ms
[CONSOLE DATA RX] Speed:    2.60 kph | dT=  1.12ms
[CONSOLE DATA RX] Speed:    1.90 kph | dT=  0.60ms
[CONSOLE DATA RX] Speed:  162.40 kph | dT=  1.14ms
[CONSOLE DATA RX] Speed:  199.50 kph | dT=  1.15ms
[CONSOLE DATA RX] Speed:  119.80 kph | dT=  0.64ms
[CONSOLE DATA RX] Speed:  184.30 kph | dT=  1.16ms
[CONSOLE DATA RX] Speed:   18.90 kph | dT=  1.19ms
[CONSOLE DATA RX] Speed:  171.50 kph | dT=  0.80ms
[CONSOLE DATA RX] Speed:   77.00 kph | dT=  1.29ms
[CONSOLE DATA RX] Speed:   40.00 kph | dT=  0.69ms
[CONSOLE DATA RX] Speed:   74.10 kph | dT=  1.25ms
[CONSOLE DATA RX] Speed:   48.20 kph | dT=  0.97ms
[CONSOLE DATA RX] Speed:   78.90 kph | dT=  0.66ms
[CONSOLE DATA RX] Speed:   18.20 kph | dT=  1.15ms
[CONSOLE DATA RX] Speed:  116.50 kph | dT=  1.19ms
[CONSOLE DATA RX] Speed:   50.00 kph | dT=  0.70ms
[CONSOLE DATA RX] Speed:   84.30 kph | dT=  1.22ms
[CONSOLE DATA RX] Speed:   83.00 kph | dT=  1.22ms
[CONSOLE DATA RX] Speed:   75.20 kph | dT=  0.61ms
[CONSOLE DATA RX] Speed:  135.40 kph | dT=  1.16ms
[CONSOLE DATA RX] Speed:  188.60 kph | dT=  1.19ms
[CONSOLE DATA RX] Speed:   52.70 kph | dT=  0.62ms
[CONSOLE DATA RX] Speed:  170.70 kph | dT=  1.17ms
```

```
[INFO - GUI] Application GUI Initialized.
[INFO - GUI] Successfully bound to UDP port 50000 for receiving.
[INFO - GUI] Data receiver thread starting (data will appear in console).
[INFO - GUI] Data receiver active. Listening on port 50000 for data.
[CMD TX - GUI] Sent command: 0x00 to 169.254.32.100:50000
[CMD TX - GUI] Sent command: 0x01 to 169.254.32.100:50000
[DATA RX - GUI] ACK for SubCmd=0x00, Status=OK | dT=188480.66ms
[DATA RX - GUI] NACK from Slave (Master Poll Resp or Frame Parse Fail) | dT=  8.21ms
[DATA RX - GUI] ACK for SubCmd=0x01, Status=OK | dT=  0.04ms
```

```
[CONSOLE DATA RX] Gyro(raw): X=-10286 Y= 16055 Z=-
[CONSOLE DATA RX] Gyro(raw): X=  3935 Y= 15386 Z=
[CONSOLE DATA RX] Gyro(raw): X= 10488 Y= -2503 Z=-
[CONSOLE DATA RX] Gyro(raw): X=-19903 Y=-15055 Z=
[CONSOLE DATA RX] Gyro(raw): X=-18567 Y=-16771 Z=
[CONSOLE DATA RX] Gyro(raw): X= -3751 Y=  6783 Z=-
[CONSOLE DATA RX] Gyro(raw): X=   943 Y=-15899 Z=
[CONSOLE DATA RX] Gyro(raw): X= -9430 Y=-17019 Z=
[CONSOLE DATA RX] Gyro(raw): X=  9562 Y= -8712 Z=
[CONSOLE DATA RX] Gyro(raw): X=  5626 Y= -4427 Z=
[CONSOLE DATA RX] Gyro(raw): X= 11664 Y= -2899 Z=-
[CONSOLE DATA RX] Gyro(raw): X= 11737 Y=  4148 Z=
[CONSOLE DATA RX] Gyro(raw): X= -4092 Y= -9892 Z=-
[CONSOLE DATA RX] Gyro(raw): X= -2019 Y= 13839 Z=
[CONSOLE DATA RX] Gyro(raw): X=-13395 Y= 19902 Z=
[CONSOLE DATA RX] Gyro(raw): X=-13678 Y=-19186 Z=
[CONSOLE DATA RX] Gyro(raw): X= -5538 Y= 19326 Z=-
[CONSOLE DATA RX] Gyro(raw): X=  4157 Y= -3881 Z=
[CONSOLE DATA RX] Gyro(raw): X=  1506 Y= 19155 Z=  -732 | dT=  1.20ms
[CONSOLE DATA RX] Gyro(raw): X=  9726 Y=  6260 Z=-19379 | dT=  0.63ms
[CONSOLE DATA RX] Gyro(raw): X= 19403 Y=  2860 Z= 10790 | dT=  1.17ms
[CONSOLE DATA RX] Gyro(raw): X=-14998 Y=  5343 Z= -1723 | dT=  1.13ms
[CONSOLE DATA RX] Gyro(raw): X=  3627 Y= -7788 Z= 17718 | dT=  1.18ms
[CONSOLE DATA RX] Gyro(raw): X=  3794 Y=-17053 Z= -8176 | dT=  0.61ms
[CONSOLE DATA RX] Gyro(raw): X= -3388 Y= -6636 Z=-12955 | dT=  1.12ms
[CONSOLE DATA RX] Gyro(raw): X= 11332 Y= -2754 Z= 17517 | dT=  1.12ms
[CONSOLE DATA RX] Gyro(raw): X=-16917 Y=  2439 Z=  -599 | dT=  1.10ms
[CONSOLE DATA RX] Gyro(raw): X=-15095 Y=  9699 Z= -5972 | dT=  0.58ms
[CONSOLE DATA RX] Gyro(raw): X= -9101 Y=  -733 Z=-12648 | dT=  1.11ms
[CONSOLE DATA RX] Gyro(raw): X= -5144 Y=-13510 Z= 19359 | dT=  1.14ms
[CONSOLE DATA RX] Gyro(raw): X= -5674 Y=  3664 Z= -2047 | dT=  1.15ms
[CONSOLE DATA RX] Gyro(raw): X= -9947 Y=-15677 Z=-18468 | dT=  0.59ms
```

```
[INFO - GUI] Data receiver thread starting (data will appear in console).
[INFO - GUI] Data receiver active. Listening on port 50000 for data.
[CMD TX - GUI] Sent command: 0x00 to 169.254.32.100:50000
[CMD TX - GUI] Sent command: 0x01 to 169.254.32.100:50000
[DATA RX - GUI] ACK for SubCmd=0x00, Status=OK | dT=188480.66ms
[DATA RX - GUI] NACK from Slave (Master Poll Resp or Frame Parse Fail) | dT=  8.21ms
[DATA RX - GUI] ACK for SubCmd=0x01, Status=OK | dT=  0.04ms
[CMD TX - GUI] Sent command: 0x02 to 169.254.32.100:50000
[DATA RX - GUI] ACK for SubCmd=0x02, Status=OK | dT=  0.09ms
```



```
[CONSOLE DATA RX] Accel(g): X=-2.540 Y=-2.840 Z= 7
[CONSOLE DATA RX] Accel(g): X= 0.240 Y= 2.270 Z= 6
[CONSOLE DATA RX] Accel(g): X=-0.890 Y=-2.080 Z= 6
[CONSOLE DATA RX] Accel(g): X=-0.510 Y=-3.120 Z= 4
[CONSOLE DATA RX] Accel(g): X=-1.220 Y=-2.450 Z= 5
[CONSOLE DATA RX] Accel(g): X= 0.610 Y=-1.920 Z= 7
[CONSOLE DATA RX] Accel(g): X=-0.750 Y=-2.440 Z= 2
[CONSOLE DATA RX] Accel(g): X=-2.160 Y=-0.390 Z= 4
[CONSOLE DATA RX] Accel(g): X=-2.710 Y=-0.520 Z= 8
[CONSOLE DATA RX] Accel(g): X=-1.370 Y=-3.410 Z= 8
[CONSOLE DATA RX] Accel(g): X= 3.500 Y=-3.320 Z= 5
[CONSOLE DATA RX] Accel(g): X= 0.380 Y=-3.250 Z= 6
[CONSOLE DATA RX] Accel(g): X= 1.000 Y= 0.160 Z= 4
[CONSOLE DATA RX] Accel(g): X=-0.570 Y= 1.540 Z= 6
[CONSOLE DATA RX] Accel(g): X= 1.480 Y=-3.860 Z= 4
[CONSOLE DATA RX] Accel(g): X=-0.780 Y=-0.250 Z= 7
[CONSOLE DATA RX] Accel(g): X= 3.320 Y= 0.460 Z= 4
[CONSOLE DATA RX] Accel(g): X=-0.290 Y= 3.260 Z= 8.
[CONSOLE DATA RX] Accel(g): X=-3.600 Y=-2.660 Z= 1.810 | dT=  0.58ms
[CONSOLE DATA RX] Accel(g): X= 2.170 Y= 0.470 Z= 4.340 | dT=  1.21ms
[CONSOLE DATA RX] Accel(g): X= 3.590 Y= 3.230 Z= 3.260 | dT=  1.16ms
[CONSOLE DATA RX] Accel(g): X= 2.320 Y= 0.580 Z= 8.740 | dT=  0.67ms
[CONSOLE DATA RX] Accel(g): X= 1.140 Y=-3.650 Z= 3.480 | dT=  1.19ms
[CONSOLE DATA RX] Accel(g): X=-0.560 Y= 3.060 Z= 6.150 | dT=  1.11ms
[CONSOLE DATA RX] Accel(g): X= 0.640 Y= 1.420 Z= 5.510 | dT=  0.59ms
[CONSOLE DATA RX] Accel(g): X= 1.760 Y= 1.450 Z= 1.940 | dT=  1.17ms
[CONSOLE DATA RX] Accel(g): X= 0.090 Y= 2.550 Z= 3.930 | dT=  1.17ms
[CONSOLE DATA RX] Accel(g): X=-1.730 Y=-0.380 Z= 3.150 | dT=  1.17ms
[CONSOLE DATA RX] Accel(g): X=-0.320 Y= 3.590 Z= 7.190 | dT=  0.62ms
[CONSOLE DATA RX] Accel(g): X=-0.080 Y= 3.150 Z= 4.800 | dT=  1.17ms
[CONSOLE DATA RX] Accel(g): X=-3.650 Y= 2.090 Z= 6.730 | dT=  1.17ms
[CONSOLE DATA RX] Accel(g): X= 3.310 Y= 3.030 Z= 6.500 | dT=  0.63ms
```

```
[DATA RX - GUI] ACK for SubCmd=0x00, Status=OK | dT=188480.66ms
[DATA RX - GUI] NACK from Slave (Master Poll Resp or Frame Parse Fail) | dT=  8.21ms
[DATA RX - GUI] ACK for SubCmd=0x01, Status=OK | dT=  0.04ms
[CMD TX - GUI] Sent command: 0x02 to 169.254.32.100:50000
[DATA RX - GUI] ACK for SubCmd=0x02, Status=OK | dT=  0.09ms
[CMD TX - GUI] Sent command: 0x01 to 169.254.32.100:50000
[DATA RX - GUI] ACK for SubCmd=0x01, Status=OK | dT=  1.24ms
[CMD TX - GUI] Sent command: 0x03 to 169.254.32.100:50000
[DATA RX - GUI] ACK for SubCmd=0x03, Status=OK | dT=  0.57ms
```

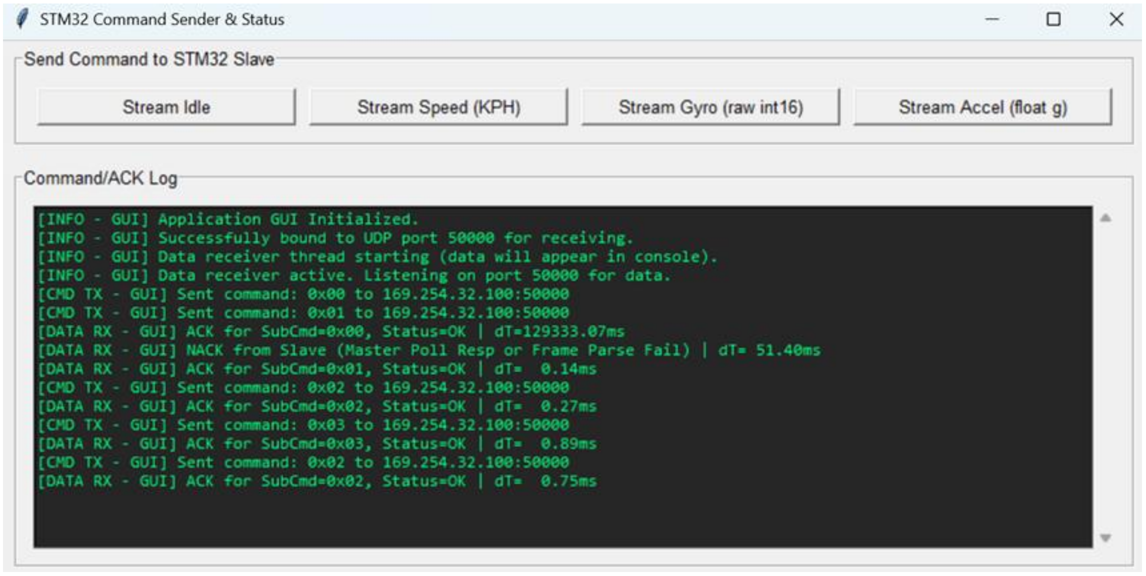
Figure 5-5: UDP Command-Response Prototype Testing Results

5.1.2 Wireshark Analysis (Integrated Prototype Tests)

Network traffic between the PC running the Python application and the STM32 Master board was captured and analyzed using Wireshark to validate the UDP communication aspects of the integrated prototype.

5.1.2.1 Packet structure

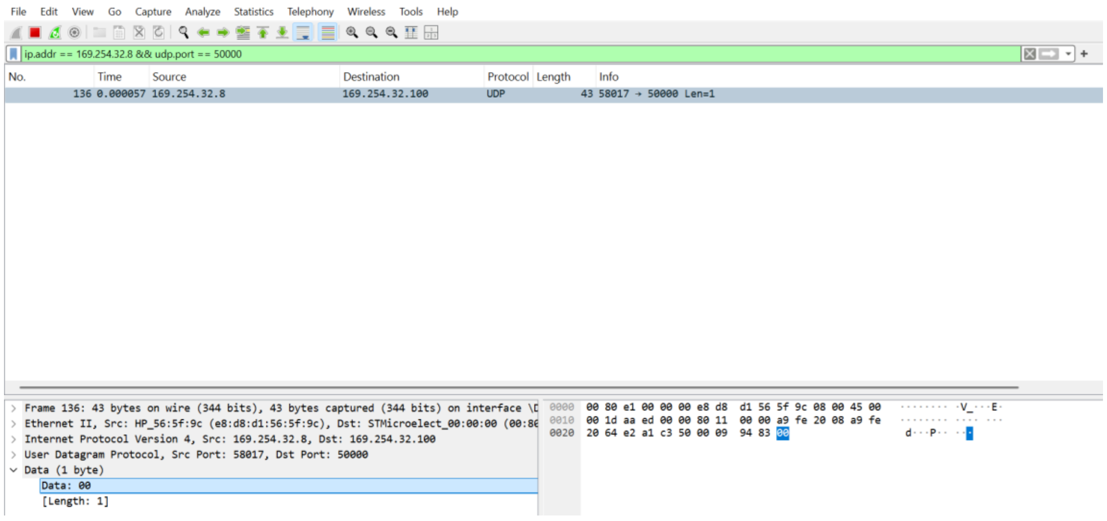
Wireshark captures confirmed that the UDP packets sent from Python script to the Master STM32 board contained 1-byte sub-commands. The sequence followed on the GUI for enabling specific data streams is shown in Figure 5-6: Python GUI sequence for Wireshark Capture demonstration.



```
[INFO - GUI] Application GUI Initialized.
[INFO - GUI] Successfully bound to UDP port 50000 for receiving.
[INFO - GUI] Data receiver thread starting (data will appear in console).
[INFO - GUI] Data receiver active. Listening on port 50000 for data.
[CMD TX - GUI] Sent command: 0x00 to 169.254.32.100:50000
[CMD TX - GUI] Sent command: 0x01 to 169.254.32.100:50000
[DATA RX - GUI] ACK for SubCmd=0x00, Status=OK | dT=129333.07ms
[DATA RX - GUI] NACK from Slave (Master Poll Resp or Frame Parse Fail) | dT= 51.40ms
[DATA RX - GUI] ACK for SubCmd=0x01, Status=OK | dT=  0.14ms
[CMD TX - GUI] Sent command: 0x02 to 169.254.32.100:50000
[DATA RX - GUI] ACK for SubCmd=0x02, Status=OK | dT=  0.27ms
[CMD TX - GUI] Sent command: 0x03 to 169.254.32.100:50000
[DATA RX - GUI] ACK for SubCmd=0x03, Status=OK | dT=  0.89ms
[CMD TX - GUI] Sent command: 0x02 to 169.254.32.100:50000
[DATA RX - GUI] ACK for SubCmd=0x02, Status=OK | dT=  0.75ms
```

Figure 5-6: Python GUI sequence for Wireshark Capture demonstration

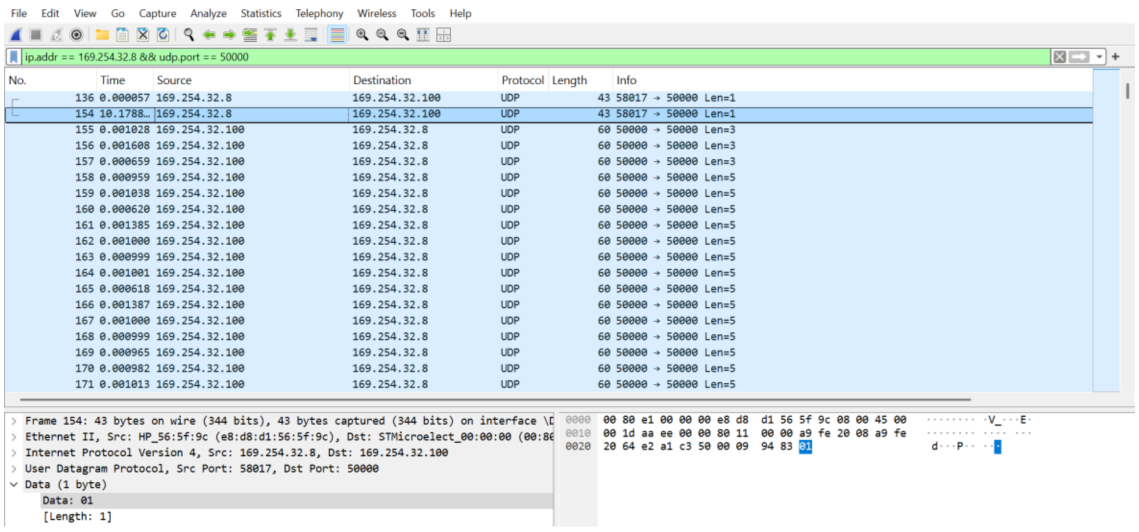
The first command sent via GUI is "Stream Idle" having length = 1 and 0x00 as its payload, and it can be seen in Figure 5-7: Wireshark - GUI sending Stream Idle.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

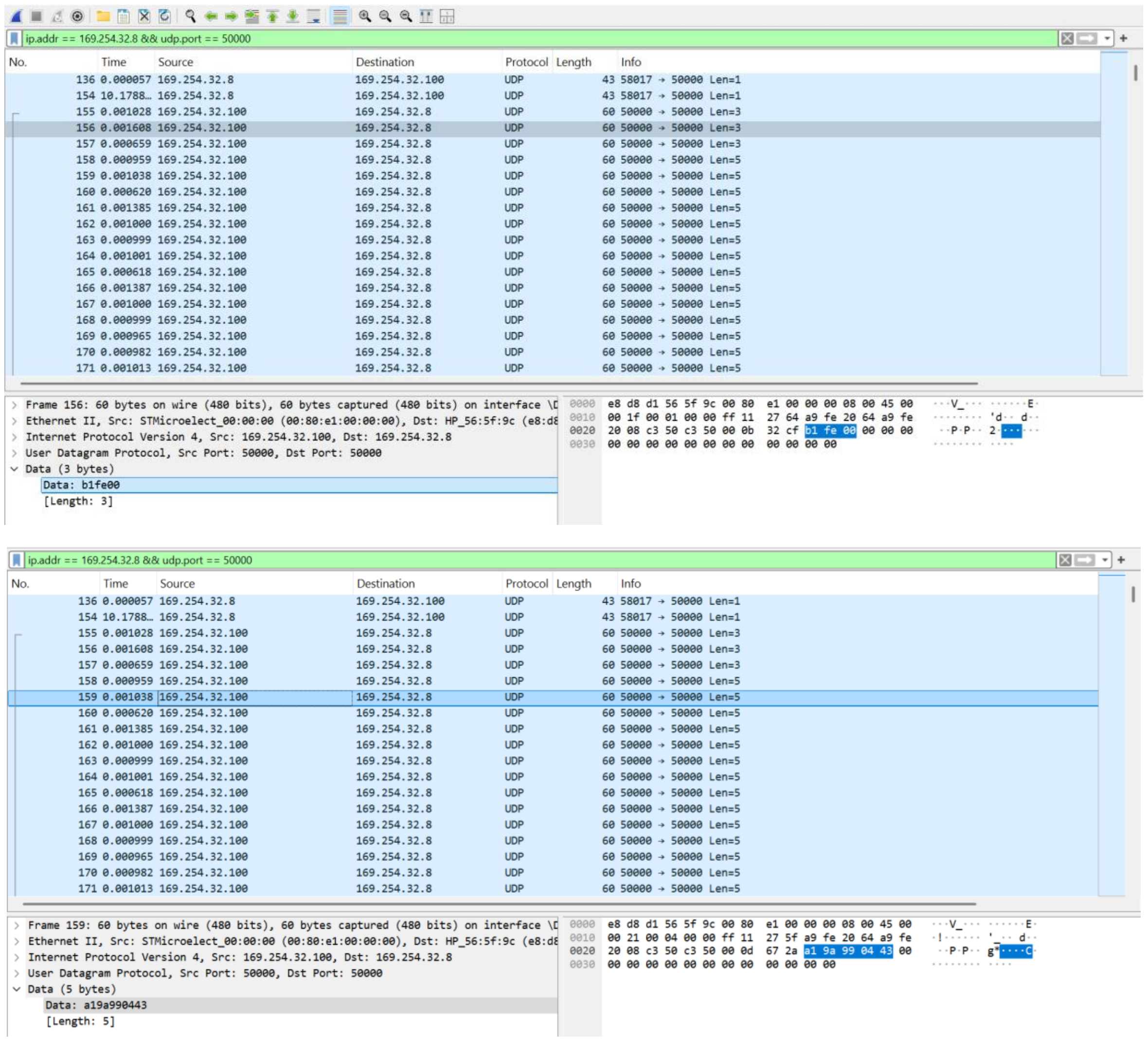
```
ip.addr == 169.254.32.8 && udp.port == 50000

No.     Time     Source          Destination      Protocol Length Info
    136 0.000057 169.254.32.8    169.254.32.100   UDP          43 58017 → 50000 Len=1

> Frame 136: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface \D
> Ethernet II, Src: HP_56:5f:9c (e8:d8:d1:56:5f:9c), Dst: STMicroelect_00:00:00 (00:8
> Internet Protocol Version 4, Src: 169.254.32.8, Dst: 169.254.32.100
> User Datagram Protocol, Src Port: 58017, Dst Port: 50000
v Data (1 byte)
    Data: 00
    [Length: 1]

0000  00 80 e1 00 00 00 e8 d8  d1 56 5f 9c 08 00 45 00   ········ ·V_···E·
0010  00 1d aa ed 00 00 80 11  00 00 a9 fe 20 08 a9 fe   ········ ···· ···
0020  20 64 e2 a1 c3 50 00 09  94 83 00                   d···P·· ··
```

Figure 5-7: Wireshark - GUI sending Stream Idle

The next command sent was "Stream Speed," having length = 1 and 0x01 as its payload. Figure 5-8: Wireshark - GUI sending Stream Speed demonstrates the pattern followed once the command is sent via GUI. The top portion of the figure shows the command sent having data = 0x01 and a length of 1. The center part of the figure shows ACK/NACK: 3 bytes total UDP payload ([SlaveFrameType (0xB0/0xB1), SubCmdEcho, Status]). The bottom part of the figure shows speed data having a UDP payload of 5 bytes ([SlaveFrameType (0xA1), float_kph]).


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
ip.addr == 169.254.32.8 && udp.port == 50000

No.     Time     Source          Destination      Protocol Length Info
    136 0.000057 169.254.32.8    169.254.32.100   UDP          43 58017 → 50000 Len=1
    154 10.1788… 169.254.32.8    169.254.32.100   UDP          43 58017 → 50000 Len=1
    155 0.001028 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=3
    156 0.001608 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=3
    157 0.000659 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=3
    158 0.000959 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    159 0.001038 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    160 0.000620 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    161 0.001385 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    162 0.001000 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    163 0.000999 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    164 0.001001 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    165 0.000618 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    166 0.001387 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    167 0.001000 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    168 0.000999 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    169 0.000965 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    170 0.000982 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5
    171 0.001013 169.254.32.100  169.254.32.8     UDP          60 50000 → 50000 Len=5

> Frame 154: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface \D
> Ethernet II, Src: HP_56:5f:9c (e8:d8:d1:56:5f:9c), Dst: STMicroelect_00:00:00 (00:8
> Internet Protocol Version 4, Src: 169.254.32.8, Dst: 169.254.32.100
> User Datagram Protocol, Src Port: 58017, Dst Port: 50000
v Data (1 byte)
    Data: 01
    [Length: 1]

0000  00 80 e1 00 00 00 e8 d8  d1 56 5f 9c 08 00 45 00   ········ ·V_···E·
0010  00 1d aa ee 00 00 80 11  00 00 a9 fe 20 08 a9 fe   ········ ···· ···
0020  20 64 e2 a1 c3 50 00 09  94 83 01                   d···P·· ··
```

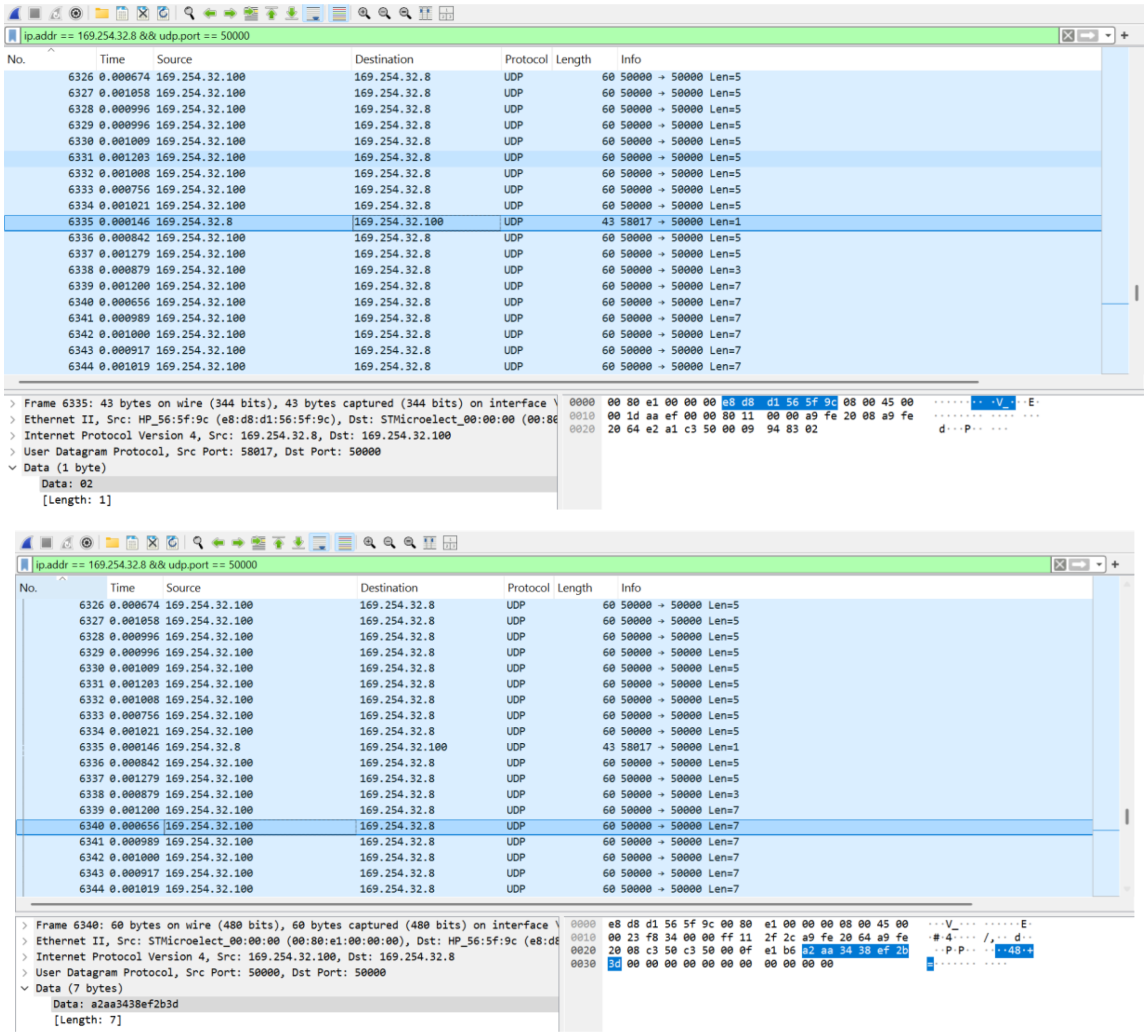


```
ip.addr == 169.254.32.8 && udp.port == 50000
No.    Time     Source          Destination     Protocol Length Info
   136 0.000057 169.254.32.8    169.254.32.100  UDP      43 58017 → 50000 Len=1
   154 10.1788… 169.254.32.8    169.254.32.100  UDP      43 58017 → 50000 Len=1
   155 0.001028 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=3
   156 0.001608 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=3
   157 0.000659 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=3
   158 0.000959 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   159 0.001038 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   160 0.000620 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   161 0.001385 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   162 0.001000 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   163 0.000999 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   164 0.001001 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   165 0.000618 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   166 0.001387 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   167 0.001000 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   168 0.000999 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   169 0.000965 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   170 0.000982 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   171 0.001013 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5

> Frame 156: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface \D
> Ethernet II, Src: STMicroelect_00:00:00 (00:80:e1:00:00:00), Dst: HP_56:5f:9c (e8:d8
> Internet Protocol Version 4, Src: 169.254.32.100, Dst: 169.254.32.8
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Data (3 bytes)
    Data: b1fe00
    [Length: 3]

0000  e8 d8 d1 56 5f 9c 00 80  e1 00 00 00 08 00 45 00   ···V_··· ······E·
0010  00 1f 00 01 00 00 ff 11  27 64 a9 fe 20 64 a9 fe   ········ 'd·· d··
0020  20 08 c3 50 c3 50 00 0b  32 cf b1 fe 00 00 00 00    ··P·P·· 2·······
0030  00 00 00 00 00 00 00 00  00 00 00 00               ········ ····
```

```
ip.addr == 169.254.32.8 && udp.port == 50000
No.    Time     Source          Destination     Protocol Length Info
   136 0.000057 169.254.32.8    169.254.32.100  UDP      43 58017 → 50000 Len=1
   154 10.1788… 169.254.32.8    169.254.32.100  UDP      43 58017 → 50000 Len=1
   155 0.001028 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=3
   156 0.001608 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=3
   157 0.000659 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=3
   158 0.000959 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   159 0.001038 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   160 0.000620 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   161 0.001385 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   162 0.001000 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   163 0.000999 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   164 0.001001 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   165 0.000618 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   166 0.001387 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   167 0.001000 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   168 0.000999 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   169 0.000965 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   170 0.000982 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
   171 0.001013 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5

> Frame 159: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface \D
> Ethernet II, Src: STMicroelect_00:00:00 (00:80:e1:00:00:00), Dst: HP_56:5f:9c (e8:d8
> Internet Protocol Version 4, Src: 169.254.32.100, Dst: 169.254.32.8
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Data (5 bytes)
    Data: a19a990443
    [Length: 5]

0000  e8 d8 d1 56 5f 9c 00 80  e1 00 00 00 08 00 45 00   ···V_··· ······E·
0010  00 21 00 04 00 00 ff 11  27 5f a9 fe 20 64 a9 fe   ·!······ '_·· d··
0020  20 08 c3 50 c3 50 00 0d  67 2a a1 9a 99 04 43 00    ··P·P·· g*····C·
0030  00 00 00 00 00 00 00 00  00 00 00 00               ········ ····
```

Figure 5-8: Wireshark - GUI sending Stream Speed

The next command sent was "Stream Gyro," having length = 1 and 0x02 as its payload. Figure 5-9: Wireshark - GUI sending Stream Gyro demonstrates the pattern followed once the command is sent via GUI. The top portion of the figure shows the command sent having data = 0x02 and a length of 1. The bottom part of the figure shows gyro data having a UDP payload of 7-byte ([SlaveFrameType (0xA2), 3x_int16_gyro]).

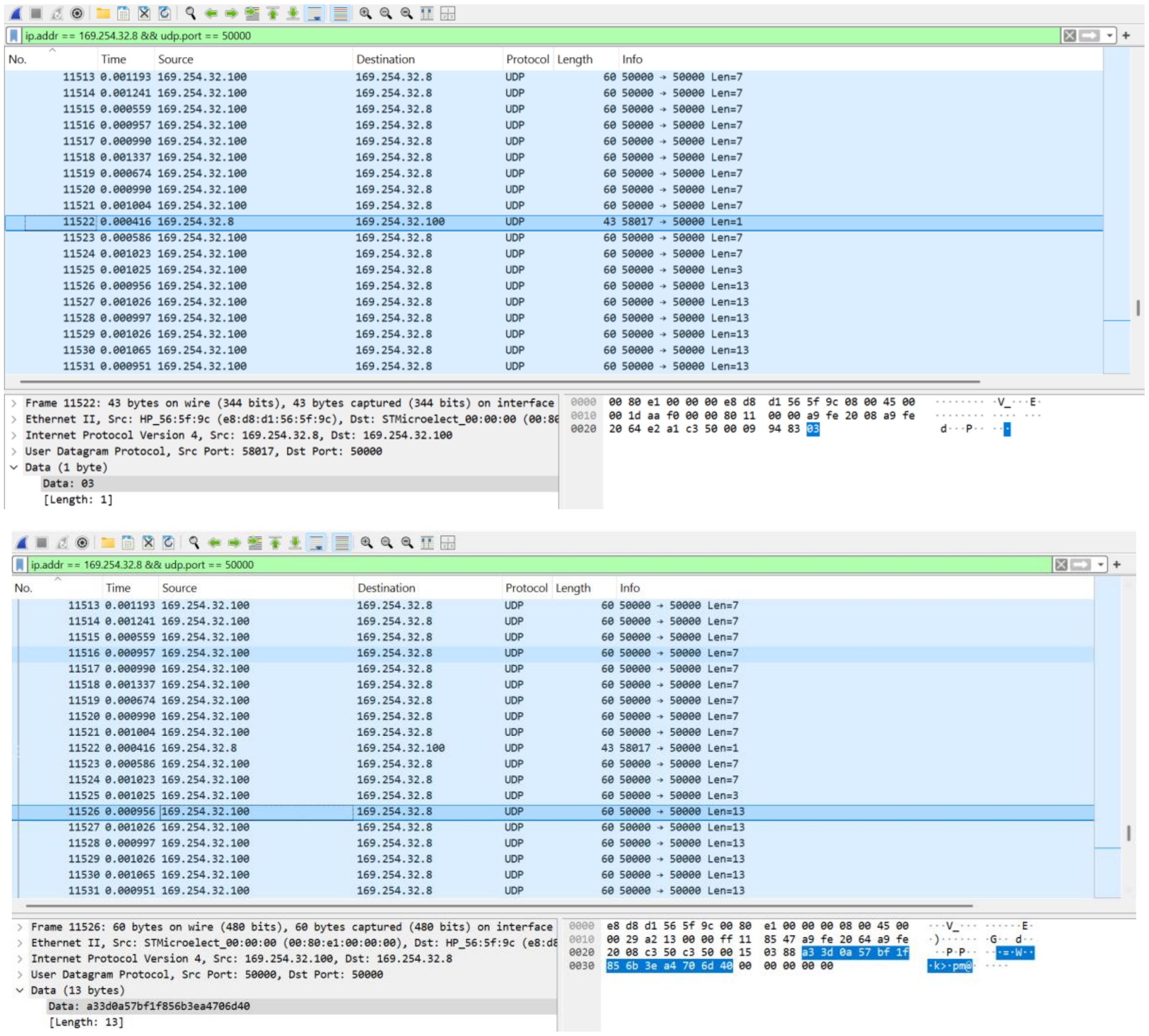


```
ip.addr == 169.254.32.8 && udp.port == 50000

No.   Time     Source          Destination     Protocol Length Info
 6326 0.000674 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6327 0.001058 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6328 0.000996 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6329 0.000996 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6330 0.001009 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6331 0.001203 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6332 0.001008 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6333 0.000756 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6334 0.001021 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6335 0.000146 169.254.32.8    169.254.32.100  UDP      43 58017 → 50000 Len=1
 6336 0.000842 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6337 0.001279 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6338 0.000879 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=3
 6339 0.001200 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
 6340 0.000656 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
 6341 0.000989 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
 6342 0.001000 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
 6343 0.000917 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
 6344 0.001019 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7

> Frame 6335: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface \
> Ethernet II, Src: HP_56:5f:9c (e8:d8:d1:56:5f:9c), Dst: STMicroelect_00:00:00 (00:8
> Internet Protocol Version 4, Src: 169.254.32.8, Dst: 169.254.32.100
> User Datagram Protocol, Src Port: 58017, Dst Port: 50000
v Data (1 byte)
    Data: 02
    [Length: 1]

0000  00 80 e1 00 00 00 e8 d8  d1 56 5f 9c 08 00 45 00   ········ ·V_···E·
0010  00 1d aa ef 00 00 80 11  00 00 a9 fe 20 08 a9 fe   ········ ···· ···
0020  20 64 e2 a1 c3 50 00 09  94 83 02                   d···P·· ···
```

```
ip.addr == 169.254.32.8 && udp.port == 50000

No.   Time     Source          Destination     Protocol Length Info
 6326 0.000674 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6327 0.001058 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6328 0.000996 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6329 0.000996 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6330 0.001009 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6331 0.001203 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6332 0.001008 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6333 0.000756 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6334 0.001021 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6335 0.000146 169.254.32.8    169.254.32.100  UDP      43 58017 → 50000 Len=1
 6336 0.000842 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6337 0.001279 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=5
 6338 0.000879 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=3
 6339 0.001200 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
 6340 0.000656 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
 6341 0.000989 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
 6342 0.001000 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
 6343 0.000917 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
 6344 0.001019 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7

> Frame 6340: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface \
> Ethernet II, Src: STMicroelect_00:00:00 (00:80:e1:00:00:00), Dst: HP_56:5f:9c (e8:d8
> Internet Protocol Version 4, Src: 169.254.32.100, Dst: 169.254.32.8
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Data (7 bytes)
    Data: a2aa3438ef2b3d
    [Length: 7]

0000  e8 d8 d1 56 5f 9c 00 80  e1 00 00 00 08 00 45 00   ···V_··· ······E·
0010  00 23 f8 34 00 00 ff 11  2f 2c a9 fe 20 64 a9 fe   ·#·4···· /,·· d··
0020  20 08 c3 50 c3 50 00 0f  e1 b6 a2 aa 34 38 ef 2b    ··P·P·· ····48·+
0030  3d 00 00 00 00 00 00 00  00 00 00 00               =······· ····
```

Figure 5-9: Wireshark - GUI sending Stream Gyro

The next command sent was “Stream Accel,” having length = 1 and 0x03 as its payload. Figure 5-10: Wireshark - GUI sending Stream Accel demonstrates the pattern followed once the command is sent via GUI. The top portion of the figure shows the command sent having data = 0x03 and a length of 1. The bottom part of the figure shows acceleration data having a UDP payload of 13-byte ([SlaveFrameType (0xA3), 3x_float_accel]).

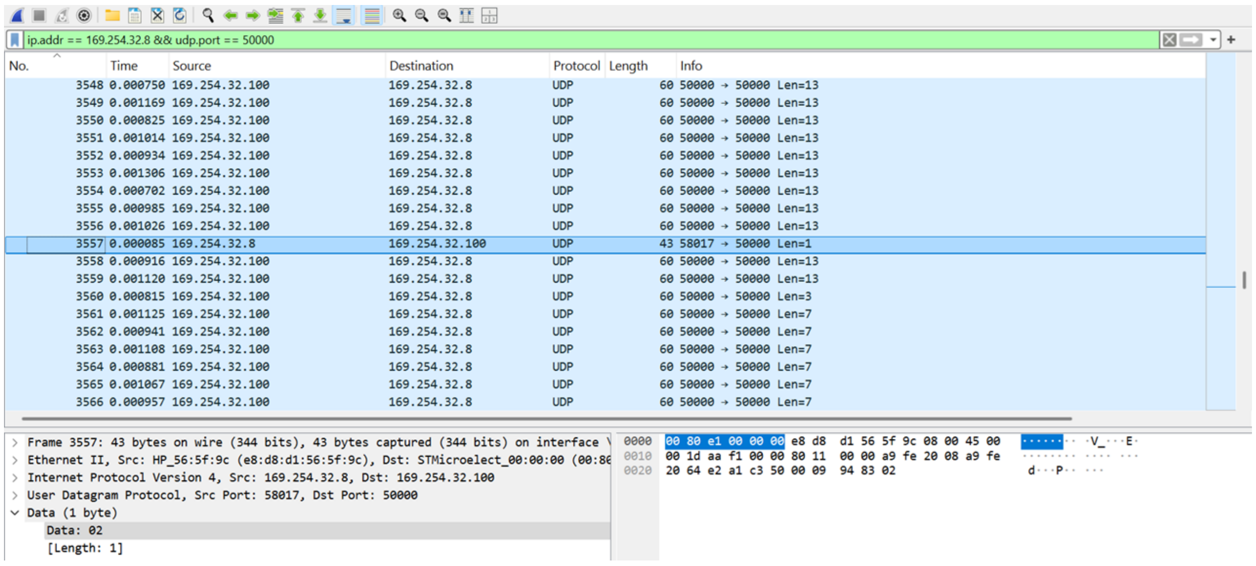


```
ip.addr == 169.254.32.8 && udp.port == 50000

No.    Time     Source          Destination     Protocol Length Info
11513 0.001193 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11514 0.001241 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11515 0.000559 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11516 0.000957 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11517 0.000990 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11518 0.001337 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11519 0.000674 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11520 0.000990 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11521 0.001004 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11522 0.000416 169.254.32.8    169.254.32.100  UDP      43 58017 → 50000 Len=1
11523 0.000586 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11524 0.001023 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11525 0.001025 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=3
11526 0.000956 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13
11527 0.001026 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13
11528 0.000997 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13
11529 0.001026 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13
11530 0.001065 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13
11531 0.000951 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13

> Frame 11522: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface
> Ethernet II, Src: HP_56:5f:9c (e8:d8:d1:56:5f:9c), Dst: STMicroelect_00:00:00 (00:8
> Internet Protocol Version 4, Src: 169.254.32.8, Dst: 169.254.32.100
> User Datagram Protocol, Src Port: 58017, Dst Port: 50000
v Data (1 byte)
    Data: 03
    [Length: 1]

0000  00 80 e1 00 00 00 e8 d8  d1 56 5f 9c 08 00 45 00   ········ ·V_···E·
0010  00 1d aa f0 00 00 80 11  00 00 a9 fe 20 08 a9 fe   ········ ···· ···
0020  20 64 e2 a1 c3 50 00 09  94 83 03                   d···P·· ···
```

```
ip.addr == 169.254.32.8 && udp.port == 50000

No.    Time     Source          Destination     Protocol Length Info
11513 0.001193 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11514 0.001241 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11515 0.000559 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11516 0.000957 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11517 0.000990 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11518 0.001337 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11519 0.000674 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11520 0.000990 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11521 0.001004 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11522 0.000416 169.254.32.8    169.254.32.100  UDP      43 58017 → 50000 Len=1
11523 0.000586 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11524 0.001023 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=7
11525 0.001025 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=3
11526 0.000956 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13
11527 0.001026 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13
11528 0.000997 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13
11529 0.001026 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13
11530 0.001065 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13
11531 0.000951 169.254.32.100  169.254.32.8    UDP      60 50000 → 50000 Len=13

> Frame 11526: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface
> Ethernet II, Src: STMicroelect_00:00:00 (00:80:e1:00:00:00), Dst: HP_56:5f:9c (e8:d8
> Internet Protocol Version 4, Src: 169.254.32.100, Dst: 169.254.32.8
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Data (13 bytes)
    Data: a33d0a57bf1f856b3ea4706d40
    [Length: 13]

0000  e8 d8 d1 56 5f 9c 00 80  e1 00 00 00 08 00 45 00   ···V_··· ······E·
0010  00 29 a2 13 00 00 ff 11  85 47 a9 fe 20 64 a9 fe   ·)······ ·G·· d··
0020  20 08 c3 50 c3 50 00 15  03 88 a3 3d 0a 57 bf 1f    ··P·P·· ··=·W··
0030  85 6b 3e a4 70 6d 40 00  00 00 00 00               ·k>·pm@· ····
```

Figure 5-10: Wireshark - GUI sending Stream Accel

The next command sent for the verification of the Wireshark capture was "Stream Gyro," and Figure 5-11: Wireshark Capture for Command Verification shows the pattern of shifting from receiving acceleration data to receiving gyro data.

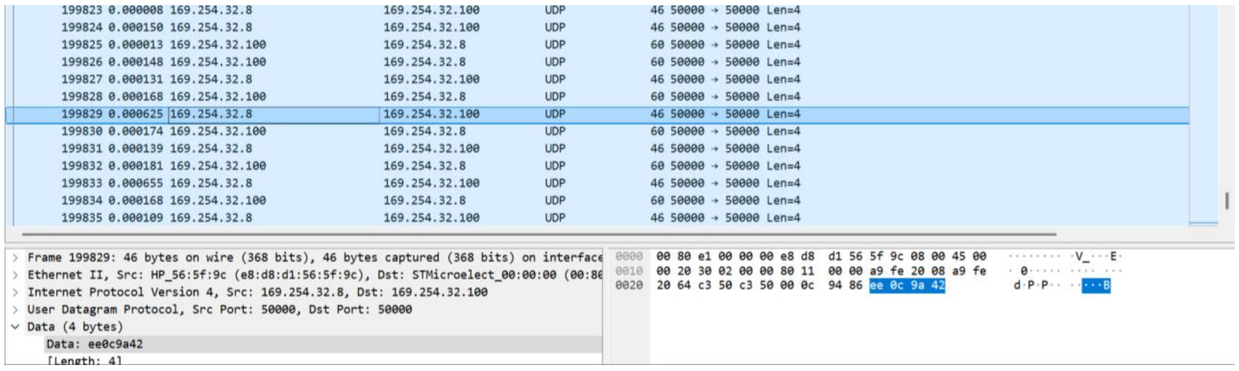


```
ip.addr == 169.254.32.8 && udp.port == 50000

No.     Time     Source           Destination      Protocol Length Info
   3548 0.000750 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3549 0.001169 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3550 0.000825 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3551 0.001014 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3552 0.000934 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3553 0.001306 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3554 0.000702 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3555 0.000985 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3556 0.001026 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3557 0.000085 169.254.32.8     169.254.32.100   UDP      43 58017 → 50000 Len=1
   3558 0.000916 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3559 0.001120 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=13
   3560 0.000815 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=3
   3561 0.001125 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=7
   3562 0.000941 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=7
   3563 0.001108 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=7
   3564 0.000881 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=7
   3565 0.001067 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=7
   3566 0.000957 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=7

> Frame 3557: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface \
> Ethernet II, Src: HP_56:5f:9c (e8:d8:d1:56:5f:9c), Dst: STMicroelect_00:00:00 (00:8€
> Internet Protocol Version 4, Src: 169.254.32.8, Dst: 169.254.32.100
> User Datagram Protocol, Src Port: 58017, Dst Port: 50000
v Data (1 byte)
    Data: 02
    [Length: 1]

0000  00 80 e1 00 00 00 e8 d8  d1 56 5f 9c 08 00 45 00   ········ ·V_···E·
0010  00 1d aa f1 00 00 80 11  00 00 a9 fe 20 08 a9 fe   ········ ···· ···
0020  20 64 e2 a1 c3 50 00 09  94 83 02                  d···P·· ···
```

Figure 5-11: Wireshark Capture for Command Verification

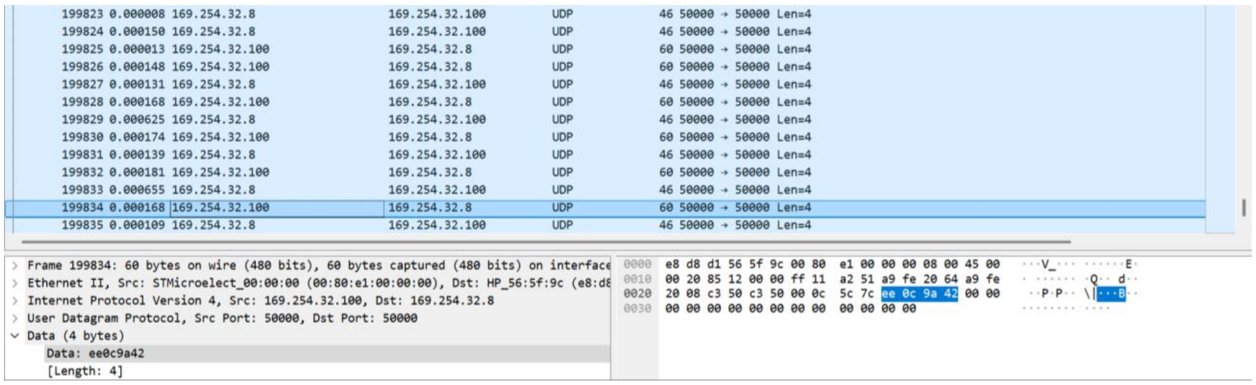
5.1.2.2 Timing Analysis

Wireshark timestamps were used to analyze the timing of UDP packet exchanges. This helped in verifying the RTTs measured by the Python application and provided in-depth analysis of the network-level latencies. Figure 5-12: Wireshark Capture of Packet Sent at 0.5ms Sending Interval shows the time taken between consecutive packets that are captured by Wireshark. The packet at capture number 199829 shows time 0.000625 seconds, and its echo is received at capture number 199835; that can be verified from Figure 5-13: Wireshark Capture of Echo Received at 0.5ms Sending Interval (Echo Test). If you add the times of all the captures in between, it sums up to around 2-ms, which is what we got on the Python script as well.



```
 199823 0.000008 169.254.32.8     169.254.32.100   UDP      46 50000 → 50000 Len=4
 199824 0.000150 169.254.32.8     169.254.32.100   UDP      46 50000 → 50000 Len=4
 199825 0.000013 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=4
 199826 0.000148 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=4
 199827 0.000131 169.254.32.8     169.254.32.100   UDP      46 50000 → 50000 Len=4
 199828 0.000168 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=4
 199829 0.000625 169.254.32.8     169.254.32.100   UDP      46 50000 → 50000 Len=4
 199830 0.000174 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=4
 199831 0.000139 169.254.32.8     169.254.32.100   UDP      46 50000 → 50000 Len=4
 199832 0.000181 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=4
 199833 0.000655 169.254.32.8     169.254.32.100   UDP      46 50000 → 50000 Len=4
 199834 0.000168 169.254.32.100   169.254.32.8     UDP      60 50000 → 50000 Len=4
 199835 0.000109 169.254.32.8     169.254.32.100   UDP      46 50000 → 50000 Len=4

> Frame 199829: 46 bytes on wire (368 bits), 46 bytes captured (368 bits) on interface
> Ethernet II, Src: HP_56:5f:9c (e8:d8:d1:56:5f:9c), Dst: STMicroelect_00:00:00 (00:8€
> Internet Protocol Version 4, Src: 169.254.32.8, Dst: 169.254.32.100
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Data (4 bytes)
    Data: ee0c9a42
    [Length: 4]

0000  00 80 e1 00 00 00 e8 d8  d1 56 5f 9c 08 00 45 00   ········ ·V_···E·
0010  00 20 30 02 00 00 80 11  00 00 a9 fe 20 08 a9 fe   ·0······ ···· ···
0020  20 64 c3 50 c3 50 00 0c  94 86 ee 0c 9a 42         d·P·P·· ···B
```

Figure 5-12: Wireshark Capture of Packet Sent at 0.5ms Sending Interval (Echo Test)

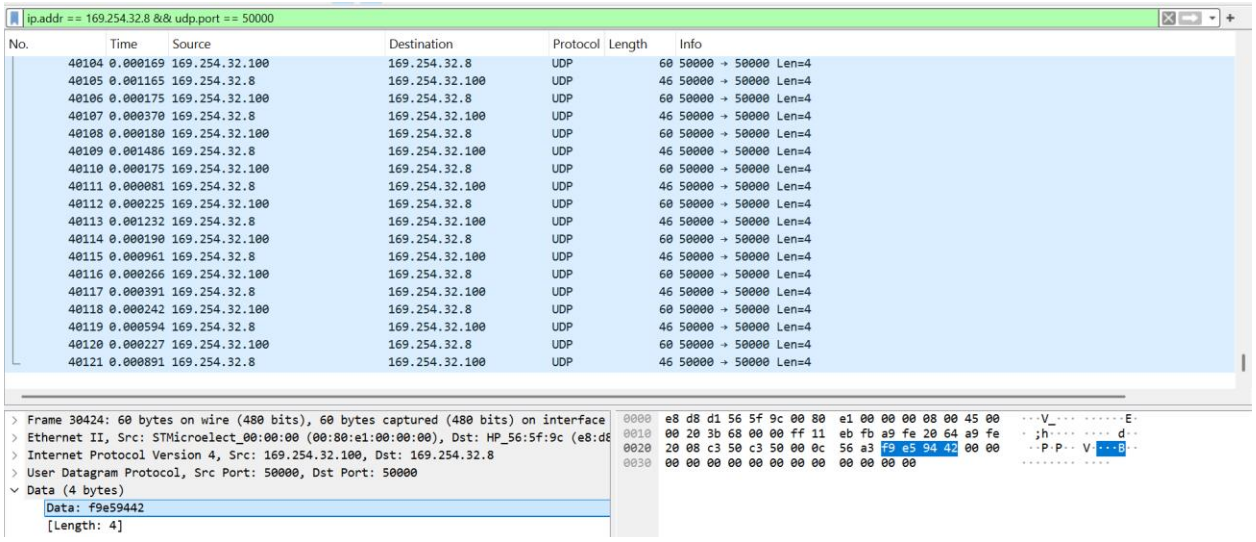


```
199823 0.000008 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
199824 0.000150 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
199825 0.000013 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
199826 0.000148 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
199827 0.000131 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
199828 0.000168 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
199829 0.000625 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
199830 0.000174 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
199831 0.000139 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
199832 0.000181 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
199833 0.000655 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
199834 0.000168 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
199835 0.000109 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4

> Frame 199834: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface
> Ethernet II, Src: STMicroelect_00:00:00 (00:80:e1:00:00:00), Dst: HP_56:5f:9c (e8:d8
> Internet Protocol Version 4, Src: 169.254.32.100, Dst: 169.254.32.8
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Data (4 bytes)
    Data: ee0c9a42
    [Length: 4]

0000  e8 d8 d1 56 5f 9c 00 80  e1 00 00 00 08 00 45 00   ···V_··· ······E·
0010  00 20 85 12 00 00 ff 11  a2 51 a9 fe 20 64 a9 fe   · ······ ·Q·· d··
0020  20 08 c3 50 c3 50 00 0c  5c 7c ee 0c 9a 42 00 00    ··P·P·· \|···B··
0030  00 00 00 00 00 00 00 00  00 00 00 00               ········ ····
```

Figure 5-13: Wireshark Capture of Echo Received at 0.5ms Sending Interval (Echo Test)

5.1.2.1 IP Address and Port Confirmation

The captures consistently showed UDP packets correctly addressed to and from the configured IP addresses (e.g., Python PC at 169.254.32.8, STM32 Master at 169.254.32.100) and UDP port (50000 for Master listening, Python listening on its bound port for responses). This can be verified from Figure 5-14: Wireshark Capture with IP and Port Filter Applied, where the filter is applied (*ip.addr == 169.254.32.8 && udp.port == 50000*) for the IP address of the PC running the Python script and the port number used. Moreover, the details of the packet selected also show the name of the source (Src: STMicroelect_00:00:00) and the destination (Dst: HP_56_5f:9c), along with the port number used (src port: 50000 and dst port: 50000).



```
ip.addr == 169.254.32.8 && udp.port == 50000

No.    Time     Source            Destination      Protocol Length Info
40104 0.000169 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
40105 0.001165 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
40106 0.000175 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
40107 0.000370 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
40108 0.000180 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
40109 0.001486 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
40110 0.000175 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
40111 0.000081 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
40112 0.000225 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
40113 0.001232 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
40114 0.000190 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
40115 0.000961 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
40116 0.000266 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
40117 0.000391 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
40118 0.000242 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
40119 0.000594 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4
40120 0.000227 169.254.32.100    169.254.32.8     UDP   60 50000 → 50000 Len=4
40121 0.000891 169.254.32.8      169.254.32.100   UDP   46 50000 → 50000 Len=4

> Frame 30424: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface
> Ethernet II, Src: STMicroelect_00:00:00 (00:80:e1:00:00:00), Dst: HP_56:5f:9c (e8:d8
> Internet Protocol Version 4, Src: 169.254.32.100, Dst: 169.254.32.8
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Data (4 bytes)
    Data: f9e59442
    [Length: 4]

0000  e8 d8 d1 56 5f 9c 00 80  e1 00 00 00 08 00 45 00   ···V_··· ······E·
0010  00 20 3b 68 00 00 ff 11  eb fb a9 fe 20 64 a9 fe   · ;h···· ···· d··
0020  20 08 c3 50 c3 50 00 0c  56 a3 f9 e5 94 42 00 00    ··P·P·· V····B··
0030  00 00 00 00 00 00 00 00  00 00 00 00               ········ ····
```

Figure 5-14: Wireshark Capture with IP and Port Filter Applied

5.2 Real-Time Simulator Testing: dSPACE ControlDesk

After the successful validation of the Ethernet-SPI Bridge with the Python-based prototype testing environment, comprehensive testing was undertaken within a Hardware-in-the-Loop (HIL) setup. The developed real-time applications were deployed and executed on the dSPACE SCALEXIO platform and were facilitated by dSPACE ControlDesk for interacting and monitoring. These HIL experiments were designed to verify a range of system capabilities, from fundamental end-to-end data echo under dynamic conditions and real-time latency measurement to the more complex command-driven emulation of multi-modal sensor data streams.

5.2.1 HIL Validation of Dynamic Speed Echo Functionality

As discussed in Section 3.3.2.2, the Simulink model "Speed_Echo_CD.slx" was designed for dynamically changing the input and verification of the echo. This model was deployed on the SCALEXIO platform. The dSPACE ControlDesk HMI layout (similar to Figure 4-13: dSPACE ControlDesk interface for the Speed Echo test, with the results depicted in Figure 5-15: dSPACE ControlDesk Interface during the Dynamic Speed Echo Test) allowed for real-time interaction.

The test methodology involved dynamically adjusting the Speed_Input value within the Simulink model using a slider instrument in dSPACE ControlDesk, with values adjusted to range from 0 to 120 km/h. This input was converted to a byte array by the *PackFloatToBytes* MATLAB function and transmitted via UDP to the STM32 Master. The Master relayed this to the Slave via SPI; the Slave echoed the data back via SPI; and the Master then forwarded the final echo via UDP to the SCALEXIO application. The echoed speed, unpacked by the *UnpackBytesToFloat* function, was displayed in dSPACE ControlDesk as "*Speed_Echo/In1*".

Figure 5-15: dSPACE ControlDesk Interface during the Dynamic Speed Echo Test illustrates the interface during this dynamic echo test. The right-hand graph (green trace) plots the *Speed_Input* value as it was varied over time using the dSPACE ControlDesk slider instrument, while the left-hand graph (red trace) plots the corresponding *Speed_Echo/In1* value received back from the STM32 system. A relation between *Speed_input* and *Speed_Echo* is observed, where the echoed speed (red trace) follows the trend of dynamically changing input speed (green trace) very closely. The numerical display instrument, assigned to the echoed speed (*Data Store\Write\In1*), provides the instantaneous value. The continuous and responsive updates of the echoed speed confirmed the functional correctness of the data path and the system's ability to handle dynamically changing inputs in the HIL environment.

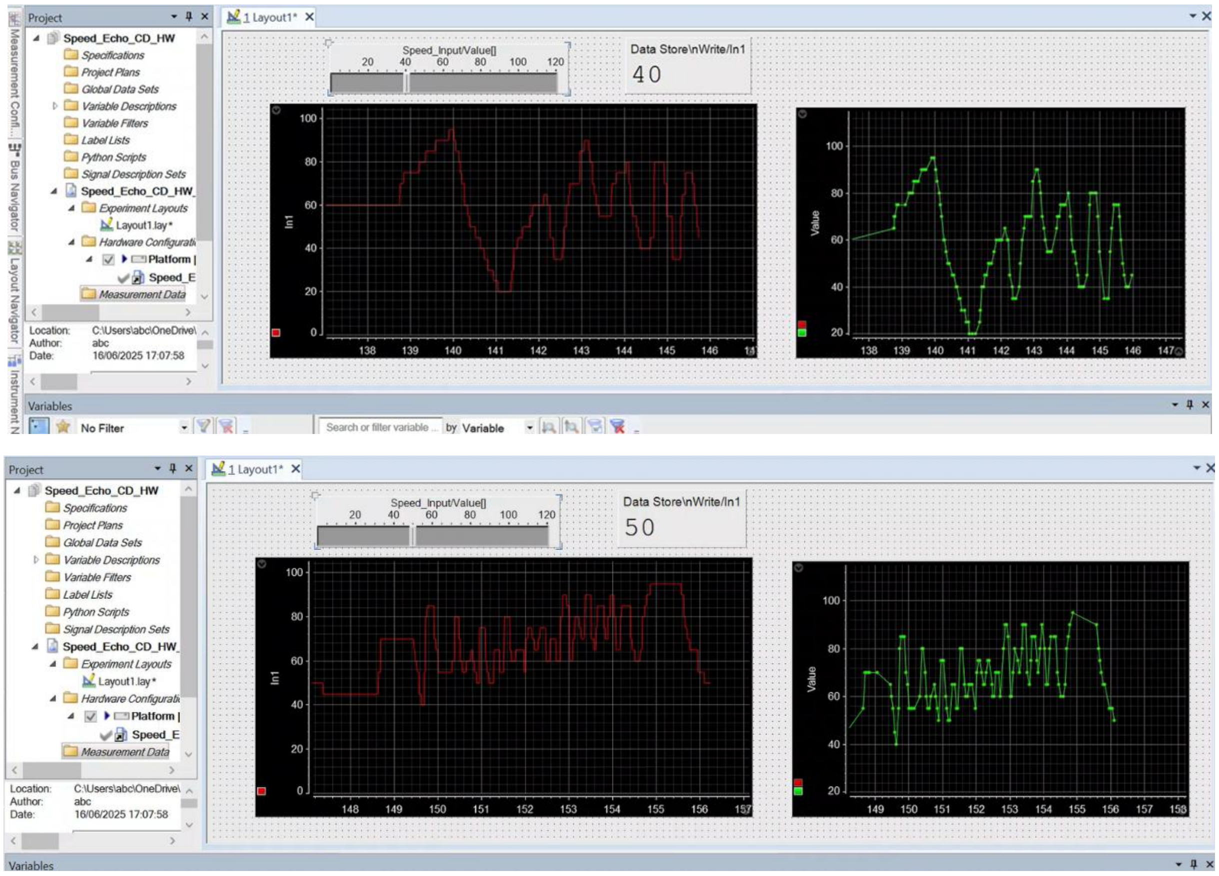
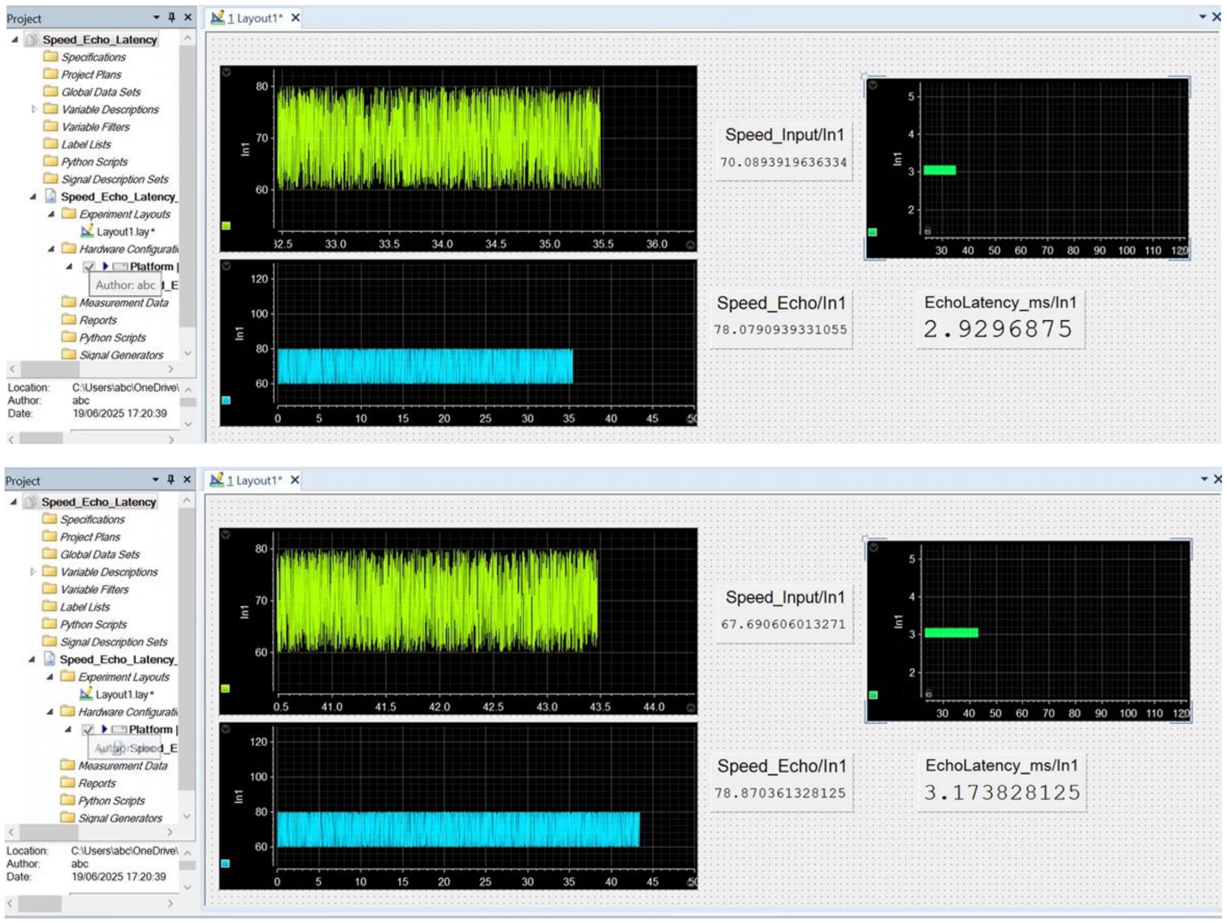


Figure 5-15: dSPACE ControlDesk Interface during the Dynamic Speed Echo Test

5.2.2 Real-Time Latency (RTT) Measurement within dSPACE Environment

As discussed in Section 3.3.2.2, the Simulink model "*Speed_Echo_Latency.slx*," incorporating the *ComputeLatencyWithBuffer* MATLAB function, was utilized to access the communication latency within the HIL setup. This model autonomously generated random speed values at 1-ms intervals, transmitted them to the STM32 system, received the echo, and calculated the RTT for each successfully matched send-receive pair.

The dSPACE ControlDesk layout for this experiment (depicted in the screenshot of Figure 5-16: dSPACE ControlDesk Interface during the Speed Echo Latency Test, focusing on the latency display) included a numerical display and a time-plot graph linked to the *EchoLatency_ms* output variable from the Simulink model. Figure 5-16: dSPACE ControlDesk Interface during the Speed Echo Latency Test shows an instance of this test, where the numerical display for *EchoLatency_ms/In1* indicates an instantaneous RTT of approximately 2.93-ms. The accompanying time-plot graph for latency (top-right in the figure) visually demonstrated the jitter in RTT values over the test duration, with fluctuations observed, for example, between approximately 2.9-ms and 3.2-ms. This direct RTT measurement within the dSPACE environment, reflecting the entire dSPACE UDP send → STM32 bridge → dSPACE UDP receive cycle, provided crucial data on the real-time performance characteristics of the system. The consistent calculation and display of RTT values confirmed the system's stability and its ability to maintain the communication loop under continuous 1-ms data exchange.



Location: C:\Users\abc\OneDrive\
Author: abc
Date: 19/06/2025 17:20:39

Speed_Input/In1 70.0893919636334
Speed_Echo/In1 78.0790939331055
EchoLatency_ms/In1 2.9296875

Speed_Input/In1 67.690606013271
Speed_Echo/In1 78.870361328125
EchoLatency_ms/In1 3.173828125

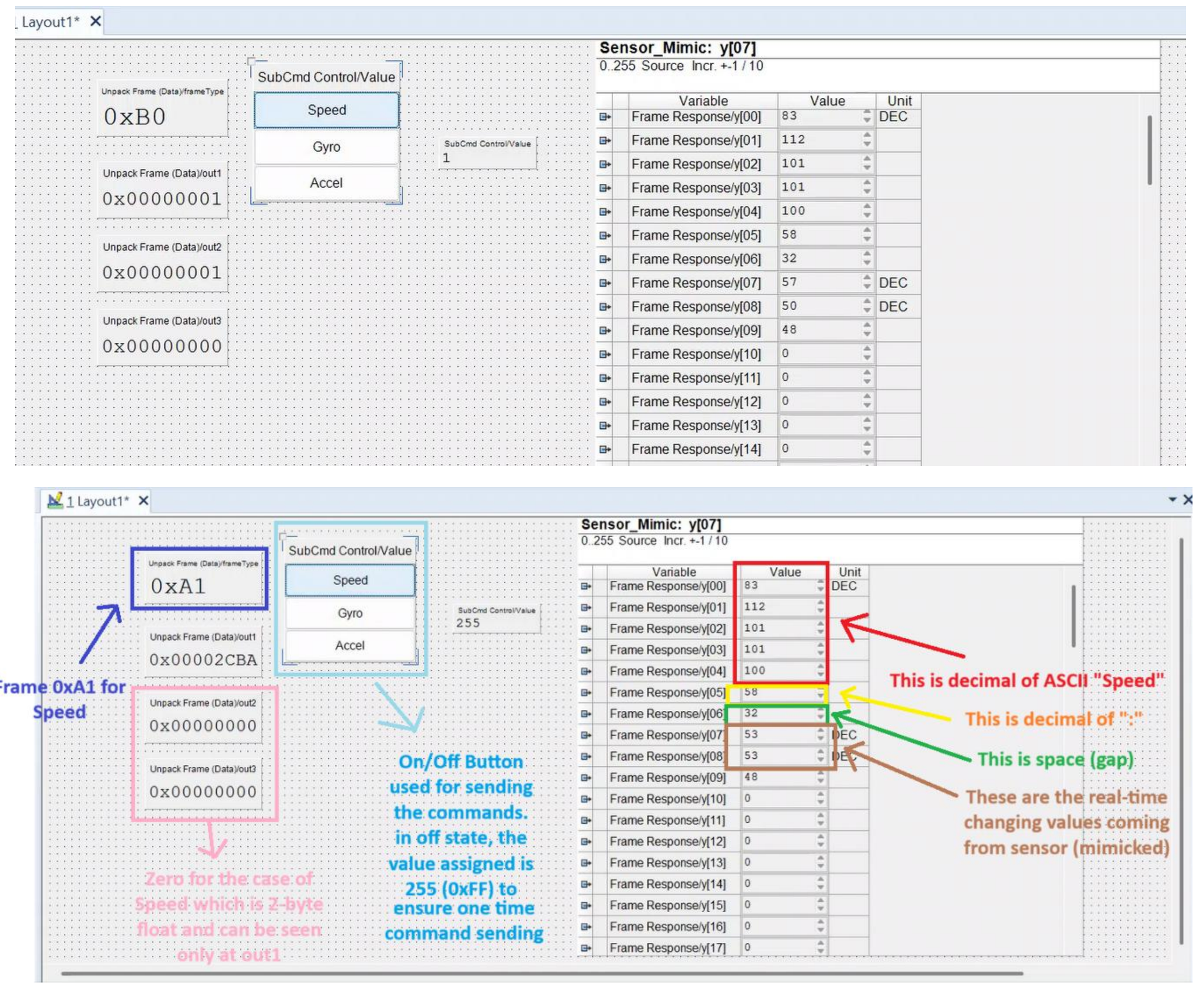
Figure 5-16: dSPACE ControlDesk Interface during the Speed Echo Latency Test

5.2.3 HIL Validation of Commanded Sensor Data Streaming

The final and most comprehensive HIL validation involved testing the "Sensor_Mimic" real-time application. The objective was to verify that the system could correctly process specific commands sent from the dSPACE ControlDesk GUI, switch the emulated sensor's data stream on the Slave board accordingly, and relay the structured sensor data back for real-time monitoring. The test procedure, as described in the section 4.4.2, used On/Off-buttons to send 1-byte sub-commands to the Slave board for "Stream Speed," "Stream Gyro," and "Stream Accel."

The results of this test confirmed the successful and robust operation of the command-driven sensor emulation. The dSPACE ControlDesk layout provided detailed insight into the received UDP packets by displaying both the raw, unpacked data from the *UnpackFrame* MATLAB function and the final formatted response array from the *FormatResponse* function.

Figure 5-17: dSPACE ControlDesk layout - Speed (Sensor_Mimic) Command and Response demonstrates the system's response after the "Stream Speed" command (0x01) was sent via the ON/OFF-button. After the reception of the ACK (*0xB0*) along with the echoed back subcommand (*Unpack Frame (Data)/out1*) and the okay status (*Unpack Frame (Data)/out2*), the numeric display *Unpack Frame (Data)/frameType* shows the value 0xA1, which correctly identifies the incoming frame as *FRAME_TYPE_DATA_VEHICLE_SPEED*. The "*out1*" numeric display shows a dynamically changing 32-bit floating-point value (represented as a hexadecimal integer in the figure), while "*out2*" and "*out3*" are zero, which, as expected, is the speed payload (single float). The *SubCmd Control/Value* numeric display shows the value 255 (0XFF) for continuous polling of the selected sensor's data. On the right side of the figure, the *FrameResponse [y[100]* array shows the ASCII output. The first few bytes correspond to the ASCII codes for "**Speed:** " (83, 112, 101, 101, 100, 58), followed by the continuously changing numerical value of the speed, further verifying that the *FormatResponse* function is working correctly.



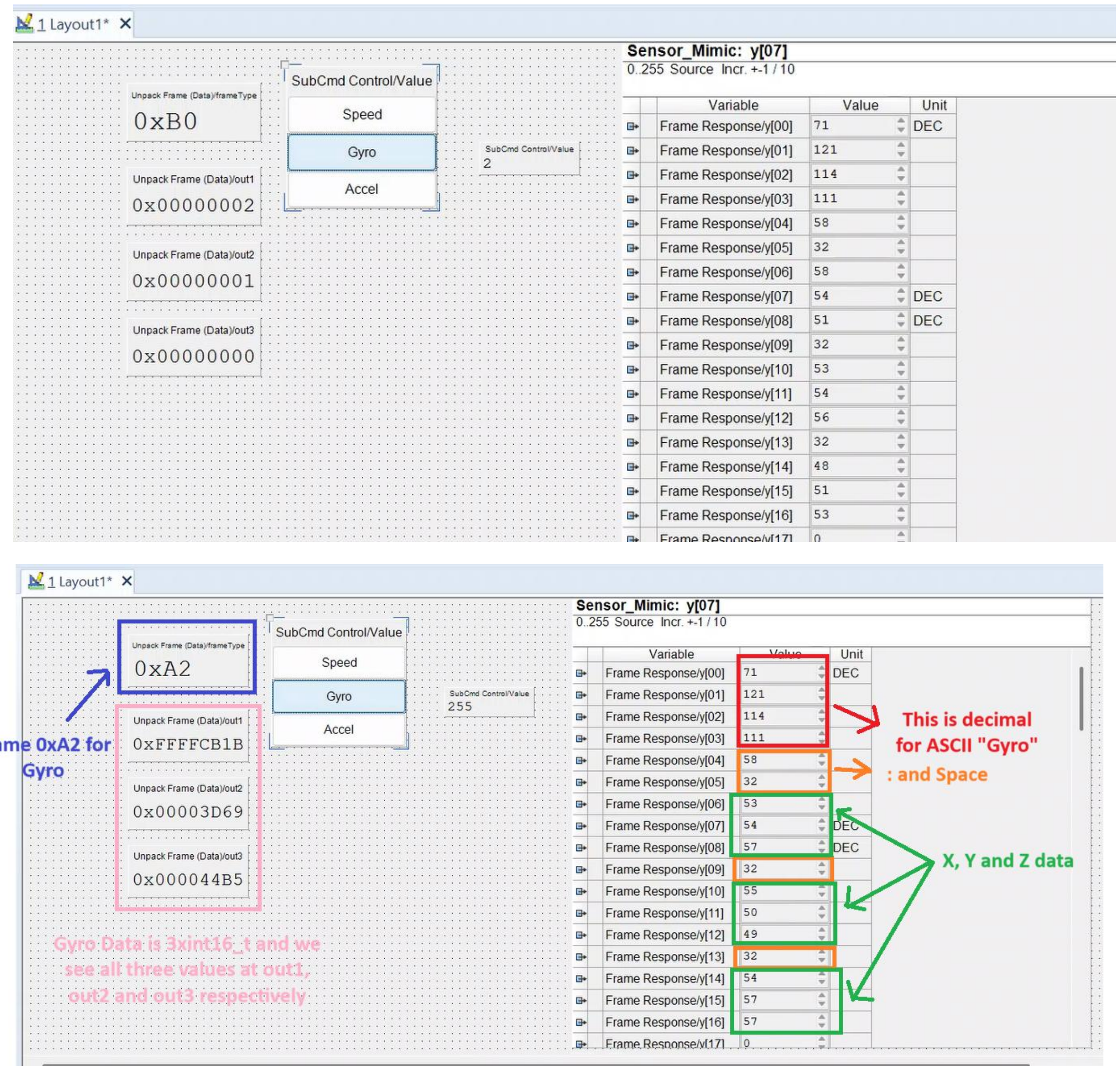
Variable	Value	Unit
Frame Response/y[00]	83	DEC
Frame Response/y[01]	112	
Frame Response/y[02]	101	
Frame Response/y[03]	101	
Frame Response/y[04]	100	
Frame Response/y[05]	58	
Frame Response/y[06]	32	
Frame Response/y[07]	57	DEC
Frame Response/y[08]	50	DEC
Frame Response/y[09]	48	
Frame Response/y[10]	0	
Frame Response/y[11]	0	
Frame Response/y[12]	0	
Frame Response/y[13]	0	
Frame Response/y[14]	0	

1 Layout1*
Unpack Frame (Data)/frameType
0xA1
Frame 0xA1 for Speed
Unpack Frame (Data)/out1
0x00002CBA
Unpack Frame (Data)/out2
0x00000000
Unpack Frame (Data)/out3
0x00000000
Zero for the case of Speed which is 2-byte float and can be seen only at out1
SubCmd Control/Value
Speed
Gyro
Accel
SubCmd Control/Value
255
On/Off Button used for sending the commands. in off state, the value assigned is 255 (0xFF) to ensure one time command sending
Sensor_Mimic: y[07]
0..255 Source Incr. +-1 / 10
This is decimal of ASCII "Speed"
This is decimal of ":"
This is space (gap)
These are the real-time changing values coming from sensor (mimicked)

Variable	Value	Unit
Frame Response/y[00]	83	DEC
Frame Response/y[01]	112	
Frame Response/y[02]	101	
Frame Response/y[03]	101	
Frame Response/y[04]	100	
Frame Response/y[05]	58	
Frame Response/y[06]	32	
Frame Response/y[07]	53	DEC
Frame Response/y[08]	53	DEC
Frame Response/y[09]	48	
Frame Response/y[10]	0	
Frame Response/y[11]	0	
Frame Response/y[12]	0	
Frame Response/y[13]	0	
Frame Response/y[14]	0	
Frame Response/y[15]	0	
Frame Response/y[16]	0	
Frame Response/y[17]	0	

Figure 5-17: dSPACE ControlDesk layout - Speed (Sensor_Mimic) Command and Response

Figure 5-18: dSPACE ControlDesk layout - Gyro (Sensor_Mimic) Command and Response shows the system's state after the "Stream Gyro" command (0x02) was sent. After successfully receiving the ACK (0xB0) and the Gyro subcommand echoed back (*Unpack Frame (Data)/out1*) along with the okay status (*Unpack Frame (Data)/out2*), the frame type display correctly shows 0xA2 (*FRAME_TYPE_DATA_IMU_GYRO*). In this case, the *UnpackFrame* function shows result on all three output ports (out1, out2, out3) with the received raw 16-bit integer values for the x, y, and z gyroscope axes. The Frame Response array on the right again displays the corresponding formatted ASCII string "Gyro: X=... Y=... Z=...".


1 Layout1*
SubCmd Control/Value
Speed
Gyro
Accel
Frame 0xA2 for Gyro
Gyro Data is 3xint16_t and we see all three values at out1, out2 and out3 respectively
This is decimal for ASCII "Gyro"
: and Space
X, Y and Z data

Unpack Frame (Data)/frameType: 0xB0
Unpack Frame (Data)/out1: 0x00000002
Unpack Frame (Data)/out2: 0x00000001
Unpack Frame (Data)/out3: 0x00000000
SubCmd Control/Value: 2

Sensor_Mimic: y[07]
0..255 Source Incr. +-1 / 10

Variable	Value	Unit
Frame Response/y[00]	71	DEC
Frame Response/y[01]	121	
Frame Response/y[02]	114	
Frame Response/y[03]	111	
Frame Response/y[04]	58	
Frame Response/y[05]	32	
Frame Response/y[06]	58	
Frame Response/y[07]	54	DEC
Frame Response/y[08]	51	DEC
Frame Response/y[09]	32	
Frame Response/y[10]	53	
Frame Response/y[11]	54	
Frame Response/y[12]	56	
Frame Response/y[13]	32	
Frame Response/y[14]	48	
Frame Response/y[15]	51	
Frame Response/y[16]	53	
Frame Response/y[17]	0	

Unpack Frame (Data)/frameType: 0xA2
Unpack Frame (Data)/out1: 0xFFFFCB1B
Unpack Frame (Data)/out2: 0x00003D69
Unpack Frame (Data)/out3: 0x000044B5
SubCmd Control/Value: 255

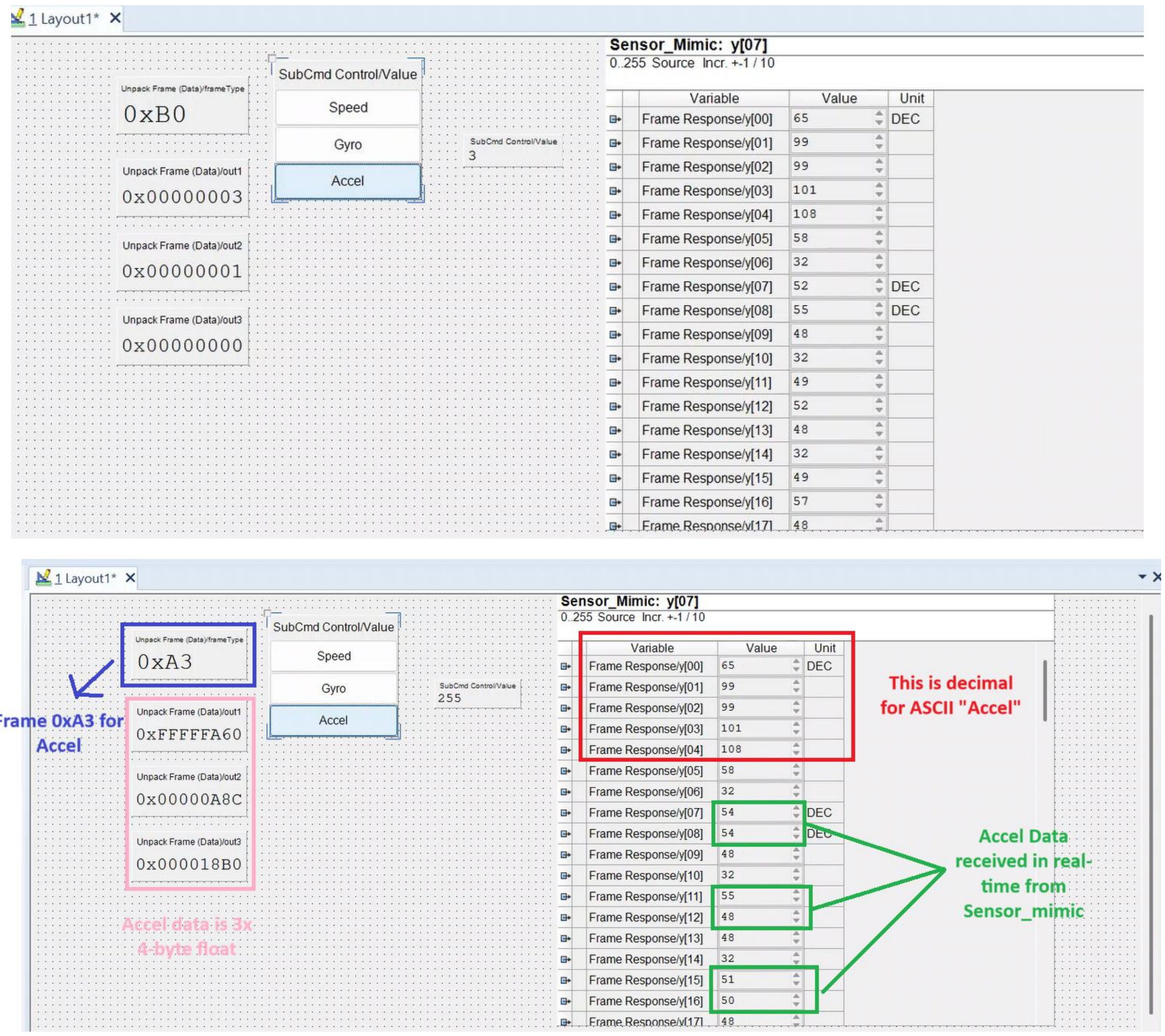
Sensor_Mimic: y[07]
0..255 Source Incr. +-1 / 10

Variable	Value	Unit
Frame Response/y[00]	71	DEC
Frame Response/y[01]	121	
Frame Response/y[02]	114	
Frame Response/y[03]	111	
Frame Response/y[04]	58	
Frame Response/y[05]	32	
Frame Response/y[06]	53	
Frame Response/y[07]	54	DEC
Frame Response/y[08]	57	DEC
Frame Response/y[09]	32	
Frame Response/y[10]	55	
Frame Response/y[11]	50	
Frame Response/y[12]	49	
Frame Response/y[13]	32	
Frame Response/y[14]	54	
Frame Response/y[15]	57	
Frame Response/y[16]	57	
Frame Response/y[17]	0	

Figure 5-18: dSPACE ControlDesk layout - Gyro (Sensor_Mimic) Command and Response

Finally, Figure 5-19: dSPACE ControlDesk layout - Accel (Sensor_Mimic) Command and Response depicts the response to the "Stream Accel" command (0x03). The frame type is identified as 0xA3 (*FRAME_TYPE_DATA_IMU_ACCEL*). Similar to the gyroscope data, the three out ports are populated with the raw 32-bit floating-point values for the x, y, and z accelerometer axes. The

Frame Response array reflects this data in its formatted string output. In all test cases, the system demonstrated the ability to seamlessly switch between data streams upon command from the dSPACE ControlDesk GUI and to correctly parse, process, and display the corresponding structured data packets received from the Slave sensor via the Ethernet-SPI Bridge. The continuous and correct updating of these values confirmed the functional integrity and real-time responsiveness of the complete sensor emulation system.


1 Layout1*
SubCmd Control/Value
Speed
Gyro
Accel
Frame 0xA3 for Accel
Accel data is 3x 4-byte float
This is decimal for ASCII "Accel"
Accel Data received in real-time from Sensor_mimic

Unpack Frame (Data)/frameType: 0xB0
Unpack Frame (Data)/out1: 0x00000003
Unpack Frame (Data)/out2: 0x00000001
Unpack Frame (Data)/out3: 0x00000000
SubCmd Control/Value: 3

Sensor_Mimic: y[07]
0..255 Source Incr. +-1 / 10

Variable	Value	Unit
Frame Response/y[00]	65	DEC
Frame Response/y[01]	99	
Frame Response/y[02]	99	
Frame Response/y[03]	101	
Frame Response/y[04]	108	
Frame Response/y[05]	58	
Frame Response/y[06]	32	
Frame Response/y[07]	52	DEC
Frame Response/y[08]	55	DEC
Frame Response/y[09]	48	
Frame Response/y[10]	32	
Frame Response/y[11]	49	
Frame Response/y[12]	52	
Frame Response/y[13]	48	
Frame Response/y[14]	32	
Frame Response/y[15]	49	
Frame Response/y[16]	57	
Frame Response/y[17]	48	

Unpack Frame (Data)/frameType: 0xA3
Unpack Frame (Data)/out1: 0xFFFFFA60
Unpack Frame (Data)/out2: 0x00000A8C
Unpack Frame (Data)/out3: 0x000018B0
SubCmd Control/Value: 255

Sensor_Mimic: y[07]
0..255 Source Incr. +-1 / 10

Variable	Value	Unit
Frame Response/y[00]	65	DEC
Frame Response/y[01]	99	
Frame Response/y[02]	99	
Frame Response/y[03]	101	
Frame Response/y[04]	108	
Frame Response/y[05]	58	
Frame Response/y[06]	32	
Frame Response/y[07]	54	DEC
Frame Response/y[08]	54	DEC
Frame Response/y[09]	48	
Frame Response/y[10]	32	
Frame Response/y[11]	55	
Frame Response/y[12]	48	
Frame Response/y[13]	48	
Frame Response/y[14]	32	
Frame Response/y[15]	51	
Frame Response/y[16]	50	
Frame Response/y[17]	48	

Figure 5-19: dSPACE ControlDesk layout - Accel (Sensor_Mimic) Command and Response

5.3 Performance Metrics Summary and Discussion

An amalgamation of key performance metrics derived from the major Python-led prototype tests as well as from the dSPACE SCALEXIO HIL validation phase is given below. The key metrics comprised Round-Trip Time (RTT), data integrity (packet loss), system stability, and qualitative responsiveness, which are collectively utilized to characterize the real-time behavior of the implemented Ethernet-SPI Bridge.

During the Python-based prototype testing phase. RTT measurements were conducted extensively.

- Initial UDP-SPI bridge echo tests, operating at 1ms send interval, resulted in average RTTs typically in the range of 2.46 ms to 2.53 ms, with observed packet loss remaining well below 0.1% (Table 5-1). When the data transmission interval was reduced to 0.5ms, slightly higher average RTTs (around 1.64 ms to 2.76 ms) were observed with an increase in packet loss of approximately 1-2%.
- The integration of the custom SPI protocol, complete with framing and software-based CRC-8 (Table 5-2), resulted in similar average RTTs for tests conducted at a send rate of 1-ms (around 2.52 ms to 2.59 ms) and maintained low packet loss (0.02% to 0.1%). When the data transmission interval was reduced to 0.5-ms, the computational overhead associated with framing and CRC likely contributed to a more notable increase in packet loss, ranging from approximately 12% to 23%. Nevertheless, the measured RTTs for successfully transmitted and echoed packets remained low, typically around 1.72 ms to 1.95 ms.
- For the advanced prototype configuration involving commanded sensor data streaming and circular DMA on the Slave board, sensor data was streamed at near 1-ms intervals per value.

The dSPACE SCALEXIO HIL validation phase provided additional details regarding the actual time operation of the system under an industrial-grade simulation.

- Dynamic speed echo tests (explained in detail previously in 5.2.1) confirmed functional correctness and qualitative responsiveness. Echoed speed values displayed in dSPACE ControlDesk closely followed dynamically changing inputs commanded via the HMI, with updates appearing in reasonable time.
- Quantitative RTT values measured from the "Speed_Echo_Latency" Simulink model (described in 5.2.2) indicated that under steady 1ms cycles of data exchange, RTTs in the dSPACE environment typically ranged from roughly 2.9-ms to 3.2-ms, with an average of roughly 3-ms. This slight increase in average RTT compared to the Python prototype would likely be explained by the processing overheads and network stack response characteristic of the dSPACE real-time operating system and hardware platform. The observed jitter of approximately 0.2-0.3-ms is considered acceptable for many HIL applications.
- For the "Sensor_Mimic" HIL test, functional verification confirmed that the system correctly handled command-driven mode switching. The reception and display of structured data for three distinct sensor types (Speed, Gyro, Accel) were validated in real-time, as shown in Figures 5-17, 5-18, and 5-19. The system maintained continuous data flow and responsiveness while switching between these modes, affirming the robustness

of the implemented protocol and the Slave's sensor emulation logic within the dSPACE HIL environment.

Overall, the performance testing on both the dSPACE HIL board and the Python-based prototype showed consistent real-time performance of the Ethernet-SPI Bridge. Round-trip time analysis confirmed that effective processing with end-to-end latencies was suitable for 1-ms cycle operations. While the Python prototype offered extensive RTT values between 2.0-ms and 2.5-ms on average, dSPACE HIL system verifications confirmed the low-latency characteristics with measured RTTs around 3-ms, including system jitter. Packet loss remained low (<1%) for 1-ms intervals in robust configurations, though testing done at higher frequencies or with the inclusion of CRC processing highlighted performance trade-offs. The sustained data throughput and stability observed in the dSPACE HIL tests, especially for dynamic data echo and commanded sensor streaming, affirm the bridge's suitability for demanding real-time HIL applications.

Chapter 6

Conclusion and Future Work

6.1 Validation

The functional capability of the system was demonstrated through the successful implementation of the Ethernet-to-SPI Bridge on the Master STM32 board. The Master board efficiently received UDP commands from both the Python-based prototype and, most importantly, from the dSPACE SCALEXIO HIL simulator using Simulink models controlled via dSPACE ControlDesk. These commands—whether they were a simple echo request for floating-point speed values or a specific 1-byte sub-command for selecting the emulated sensor data—were correctly translated into SPI transactions. Reliable data exchange with the slave STM32 board was ensured by utilizing the customized SPI protocol that employed a robust frame (start/end bytes, length, and type fields) and a software-implemented CRC-8 checksum.

In response, the Slave board accurately sent the echo back or emulated the behavior of the multi-functional sensor. Specifically in the case of the emulated sensor, it efficiently processed the SPI commands received from the Master, and on the basis of these commands, it returned an appropriate response. These responses included sending ACK/NACK frames for control commands or streaming emulated sensor data such as vehicle speed (float), IMU gyroscope data (3x int16_t), and IMU accelerometer data (3x float). Circular DMA on the Slave board was an essential optimization, as it facilitated low-latency and continuous data handling and response generation, which was needed to obtain the desired 1-ms SPI interaction cycle time.

A key validation point was the real-time performance. The system consistently achieved the target 1-ms cycle time for SPI data exchange. Round-Trip Time (RTT) analysis, done on the Python-based prototype, indicated efficient processing, and the RTTs observed for the entire UDP-SPI-UDP loop fell within the range of 1 to 2.5 ms. The system's real-time capabilities were further validated after it was integrated with the dSPACE SCALEXIO platform. The model that was designed for dynamically controlling the speed values sent from the dSPACE ControlDesk via a slider successfully received the echoed-back value from the STM32 system and displayed it in real-time within the dSPACE ControlDesk environment. Similarly, the model created for performing command-response functionality demonstrated selection of different sensor data streams from dSPACE ControlDesk via ON/OFF button, and the corresponding emulated sensor data was received and displayed accurately. These simulations done using dSPACE tools confirmed effective command handling and timely data feedback.

Data integrity and the robustness of the system were validated through the consistent operation of the CRC-8 error detection mechanism and the implemented timeout and recovery protocols on both Master and Slave boards. The prototype testing done on the system that incorporated these mechanisms resulted in some packet loss when tested for high frequency (e.g., 0.5-ms intervals), but when tested for 1-ms intervals, the system demonstrated high success rates (>99%) even during extended runs. The successful and sustained operation within the dSPACE HIL environment further proves the stability and reliability of the designed Ethernet-SPI Bridge. This validated framework provides a practical solution for real-time sensor emulation that can meet the demanding requirements of automotive HIL testing.

6.2 Improvements

While the developed Ethernet-SPI communication bridge successfully meets its primary objectives for real-time sensor emulation in an HIL context, several avenues for potential improvements and future work have been identified. These enhancements could further increase the system's performance, robustness, flexibility, and applicability to a broader range of automotive testing scenarios.

One of the significant areas for future enhancement is master board **determinism and real-time performance optimization**. As discussed earlier, the current bare-metal implementation relies on a main polling loop for managing LwIP network events and SPI transactions. Switching the Master board's firmware from this bare-metal implementation to a Real-Time Operating System (RTOS), such as FreeRTOS, could offer potential benefits. An RTOS would enable preemptive multitasking that allows the creation of dedicated, high-priority tasks specifically for UDP packet processing and SPI interface management. This could lead to the reduction in latency jitter and improvement in the system's overall response, especially when managing simultaneous processes or moving towards more complex communication demands. Furthermore, while the current firmware implements software-based CRC-8 to ensure data integrity, it also introduces a processing overhead. In the future, STM32's integrated hardware CRC peripheral could be utilized that provides a compatible standard polynomial—or could use a custom polynomial—hence, it will offload CRC computation from the CPU and free up the resources for other critical tasks. This gives the possibility of increasing the SPI baud rate, which could lead to further throughput gain.

The **communication protocol can be enhanced with the addition of advanced features**. The current system primarily uses ACK frames from the Slave for command acknowledgment. Improving the NACK (Negative Acknowledgment) mechanism to provide more refined error codes from both the Slave (for unrecognized commands or internal errors) and the Master (for SPI transaction failures after retries) back to the HIL simulator would ensure detailed diagnostics. Moreover, providing the HIL simulator with the provision to dynamically configure SPI parameters (such as clock speed or SPI mode) on the Master board via UDP commands would offer greater flexibility when testing multiple SPI-based slave devices that have different communication requirements from each other.

Considering **system scalability**, the current design focuses on a single Master-Slave SPI link. Enhancement in the capability of the Master board's software architecture to manage communication with multiple SPI slave devices simultaneously will be a valuable addition. This would require more sophisticated NSS line management (potentially requiring additional GPIOs or an SPI NSS multiplexing scheme) and more complex command routing and data acquisition logic within the Master firmware. Acquiring and handling this variable or large amount of data from multiple sources would require proper buffer management, and processing capacity must also be considered.

Lastly, for **enhanced testing capabilities**, future work could focus on integrating the bridge with standardized automotive communication protocols. This integration can be done by translating the simulated sensor data into formats like SOME/IP (Scalable service-Oriented MiddlewarE over IP) or DDS (Data Distribution Service) before transmitting it over UDP, or by bridging with other automotive buses like CAN or LIN, which can be interfaced with the Master STM32 board physically. Developing a fault injection mechanism that allows the dSPACE environment to command the Master to introduce a specific error (e.g., flipping bits, adding time delays, dropping packets) into the SPI communication will allow for a rigorous testing of the connected slave device.

6.3 Vision

The successful implementation and testing of this Ethernet-SPI communications bridge provides an effective and functional solution for the integration of the SPI-based components with the modern, Ethernet-based HIL test setups. This work done corresponds to the future needs in the automotive industry, particularly for companies conducting large-scale HIL testing of ECUs like Battery Management Systems (BMS), advanced sensors, and actuators that commonly utilize SPI for low-latency, high-bandwidth local communication.

The deployed framework offers a cost-effective and flexible solution to solely proprietary HIL interface solutions, utilizing STM32 commercially available microcontrollers everywhere and open-source LwIP. The same underlying principles and techniques employed—custom framing, software CRC, circular DMA for high-speed data transfer, and robust error recovery—can be utilized for bridging Ethernet to any other embedded bus protocols.

Eventually, this research effectively contributes toward comprehensive HIL testing methodologies. By facilitating seamless data exchange between the simulated environment and actual SPI devices, it allows for earlier detection of integration issues, more thorough verification of component performance under realistic conditions, and faster development cycles for complex automotive systems. The foundation laid by this thesis can be extended toward developing standardized, reusable gateway modules and supporting the evolving landscape of automotive E/E architectures and ISO-compliant communication protocols.

References

[1] "Milestones:Ethernet Local Area Network (LAN), 1973-1985," [Online]. Available: https://ethw.org/Milestones:Ethernet_Local_Area_Network_(LAN),_1973-1985#Title.

[2] *IEEE Standard for Ethernet,* IEEE, 2022.

[3] "Network Reference Model," [Online]. Available: https://profinetuniversity.com/industrial-automation-ethernet/network-reference-model/.

[4] D. Smith. [Online]. Available: https://sierrahardwaredesign.com/basic-networking/what-are-the-various-types-formats-of-ethernet-frames/.

[5] "User Datagram Protocol (UDP)," [Online]. Available: https://www.geeksforgeeks.org/user-datagram-protocol-udp/.

[6] "Transport Layer Protocol (TCP)," [Online]. Available: https://www.uninets.com/blog/how-does-tcp-work.

[7] "SPI Protocol," [Online]. Available: https://www.prodigytechno.com/spi-protocol.

[8] S. Campbell, "Basics of SPI Communication," [Online]. Available: https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/.

[9] "SPI Modes," [Online]. Available: https://stackoverflow.com/questions/71351912/how-to-understand-the-spi-clock-modes.

[10] J. A. Stankovic , "Misconceptions about real-time computing: a serious problem for next-generation systems," 1988.

[11] P. A. Laplante and S. J. Ovaska, Real-Time Systems Design and Analysis, 2012.

[12] "Classification of Real-Time System," [Online]. Available: https://www.geeksforgeeks.org/real-time-systems/.

[13] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, 1997.

[14] A. Burns and A. Wellings, Real-Time Systems and Programming Languages, 2009.

[15] G. C. Buttazzo, Hard Real-Time Computing Systems, 2011.

[16] C. M. Krishna and K. G. Shin, Real-Time Systems, 1997.

[17] J. W. Valvano, Embedded Microcomputer Systems : Real Time Interfacing, 2012.

[18] R. Wilhelm, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Journals for the Design of Smart and Connected Systems,* 2008.

[19] J. W. S. Liu, "Real Time Systems," 2000.

[20] R. Isermann, J. Schaffnit and S. Sinsel, "Hardware-in-the-loop simulation for the design and testing of engine-control systems," *Control Engineering Practice,* 1999.

[21] O. Gietelink, J. Ploeg, B. De Schutter and M. Verhaegen, "Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations," *International Journal of Vehicle Mechanics and Mobility,* 2006.

[22] H. Hanselmann, "Hardware-in-the-loop simulation testing and its integration into a CACSD toolset," in *International Symposium on Computer Aided Control System Design (CACSD)*, 1996.

[23] D. Maclay, "Simulation gets into the loop," *IEEE Review,* 1997.

[24] M. Short and M. J. Pont, "Hardware in the loop simulation of embedded automotive control system," in *International Conference on Intelligent Transportation*, 2005.

[25] T. Yao, Y. Hu and L. Ding, "FPGA-Based for Implementation of Multi-Serials to Ethernet Gateway," in *International Workshop on Intelligent Systems and Applications, ISA*, 2010.

[26] S. Zoican and M. Vochin, "LwIP stack protocol for embedded sensors network," in *International Conference on Communications (COMM)*, 2012.

[27] J. Shang and H. Ding, "Application of lightweight protocol stack LwIP on embedded Ethernet," in *International Conference on Electrical and Control Engineering (ICECE)*, 2011.

[28] A. B. C. Douss, R. Abassi and D. Sauveron, "State-of-the-art survey of in-vehicle protocols and automotive Ethernet security and vulnerabilities," *Mathematical Biosciences and Engineering,* 2023.

[29] J.-L. Kuo, C.-H. Shih and Y.-C. Chen, "Performance analysis of real-time streaming under TCP and UDP in VANET via OMNET," in *International Conference on ITS Telecommunications (ITST)*, 2013.

[30] S. Karthick, S. Venkatesh, P. S. Chaithanya and G. C. S. Royal, "Implementation and Functional Verification of I2C, SPI and CAN frame protocols," in *Emerging Smart Computing and Informatics (ESCI)*, 2025.

[31] C. Zitlaw, A. Sayeed and S. L. Lim, "CRC Integration for Enhanced SPI Communication Reliability in Digital Systems," in *Multimedia University Engineering Conference (MECON)*, 2024.

[32] J. K. Lee, J. W. Ahn, J. O. Kim and S. G. Choi, "The SPI Synchronization Method for Data Efficiency and Integrity between MCUs," in *International Conference on Advanced Communication Technology (ICACT)*, 2019.

[33] "STM32 32-bit Arm Cortex MCUs," STMicroelectronics, [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html.

[34] STMicroelectronics, *STM32F2 Reference Manual - RM0033.*

[35] STMicroelectronics, *Developing applications on STM32Cube with LwIP TCP/IP stack - User Manual (UM1713).*

[36] dSPACE, *SCALEXIO Hardware Installation and Configuration Document,* 2022.

[37] STMicroelectronics, *STM32 Nucleo-144 boards (MB1137) - User Manual (1974).*

[38] STMicroelectronics, *STM32F207ZG Datasheet.*

[39] STMicroelectronics, *Description of STM32F2 HAL and low-layer drivers - User Manual (1940).*