Department of Information Engineering and Computer Science

Master's Degree in Embedded Systems

FINAL DISSERTATION

Zephyr RTOS: Design of a Deterministic Ethernet Network for Machine-to-Machine (M2M) Communications.

Supervisor
Prof. Luigi Palopoli
University of Trento

Signature:

Supervisor
Heinz-Peter Liechtenecker
KAI Kompetenzzentrum Automobil- und Industrieelektronik GmbH

Signature: Liechtenecker

Student
Sebin Shaji Philip
ID number: 219765

ACADEMIC YEAR 2021/2022

Acknowledgments

THIS thesis would not have been possible without the constant encouragement, support and help from a lot of people and I'd like to thank all of them for their kindness. First I'd like to thank my parents for their never-ending love and support.

I'd like to thank my Kompetenzzentrum Automobil- und Industrie-Elektronik (KAI) supervisor, Heinz-Peter Liechtenecker for giving me the proper guidance and support throughout the project. The constant encouragement and support provided by Benjamin Steinwender and Sascha Einspieler helped a long way for my thesis and I'd like to thank them. I'd also like to express my sincere gratitude towards the entire software and hardware team at KAI for giving me the right environment and support for conducting my project.

I'm very grateful to Prof. Luigi Palopoli, my University of Trento (UniTn) supervisor for his help and advice, both during the thesis work and during the academic study at UniTn.

I'd like to express my gratitude to the entire *Zephyr community* for the crucial support and reviews done during the driver development.

Finally I would like to thank my friends and colleagues for their support that helped me complete my thesis project successfully.

Abstract

The standard Ethernet (Institute of Electrical and Electronics Engineers (IEEE) 802.3) has become one of the primary choise of communication for industrial automation (*Industry 4.0* and *Industrial Internet of Things (IIoT)*) applications. The faster transfer rates of the high volume of data along with superior energy efficiency contribute to its popularity. However, the standard Ethernet does not provide determinism by default, which is crucial for industrial control loop applications with real-time performance requirements. Although there are several proprietary industrial Ethernet standards addressing determinism, the IEEE-based Audio Video Bridging (AVB)/Time-Sensitive Networking (TSN) standards are gaining popularity by bringing deterministic Ethernet to the IIoT. Apart from being part of the Ethernet standard, TSN brings increased security and Enterprise compatibility by bringing the costs down, making it a better fit for measurement, monitoring, and control.

In this thesis, I investigated the shortcomings and challenges of the distributed power semiconductor test setup realised by KAI [1]. Enabling a scalable Ethernet communication network along with giving deterministic guarantees for time-critical control-loop data are identified as the main challenge. The current event-driven test setup architecture is modified with a mixed Message Queuing Telemetry Transport (MQTT) and MQTT For Sensor Networks (MQTT-SN) based Machine-to-Machine (M2M) communication on Zephyr Real-Time Operating System (RTOS) to provide scalability. The *IEEE 802.1Q priority mapping* realised by the Zephyr RTOS is made used to enable deterministic M2M communication for time-critical control data, and all the results based on different Quality of Service (QoS) configurations are analysed for selecting the minimum jitter deterministic communication.

Contents

List of Figures

List of Tables

1

Introduction and Overview

This chapter explains the background and the main motivation behind the thesis work. The scope and objective of the thesis work are also defined along with the research methods adopted to carry out the work. A general outline of the entire thesis report is given at the end.

1.1 Background

Power semiconductors are widely used in complex consumer, industrial and automotive electronic devices. Modern electronic devices with ever decreasing dimensions call for the need for miniature power devices integrated into them. The increased thermal, mechanical, and electrical stress caused by the decreased size of the power devices have an adverse effect on their lifetime (e.g., the thermal degradation of power electronic components). Reliability stress testing becomes very relevant for these devices to study and model their degradation pattern and thus giving accurate lifespan predictions. The main idea behind reliability stress testing is to undergo the Device Under Test (DUT) power devices through a wide range of preconfigured temperature/power/electrical cycles and then collect and measure their varying properties periodically. The recent requirements that arise due to complex customer needs and reliability standards, a more complex distributed test setup that can accommodate a large number of DUTs were evolved (instead of a central setup with limited DUTs attached). These distributed systems with a control node attached to each DUT are connected through a communication network (e.g., Ethernet), through a centralised architecture. The existing architecture of such an advanced reliability testing setup realised by KAI as part of its "Distributed Smart Controller Network for a Modular Power Stress Test" [1] is depicted in Figure 1.1. The challenges and use-cases that arise out of this distributed communication network are described with the help of Figure 1.1 in the following section.

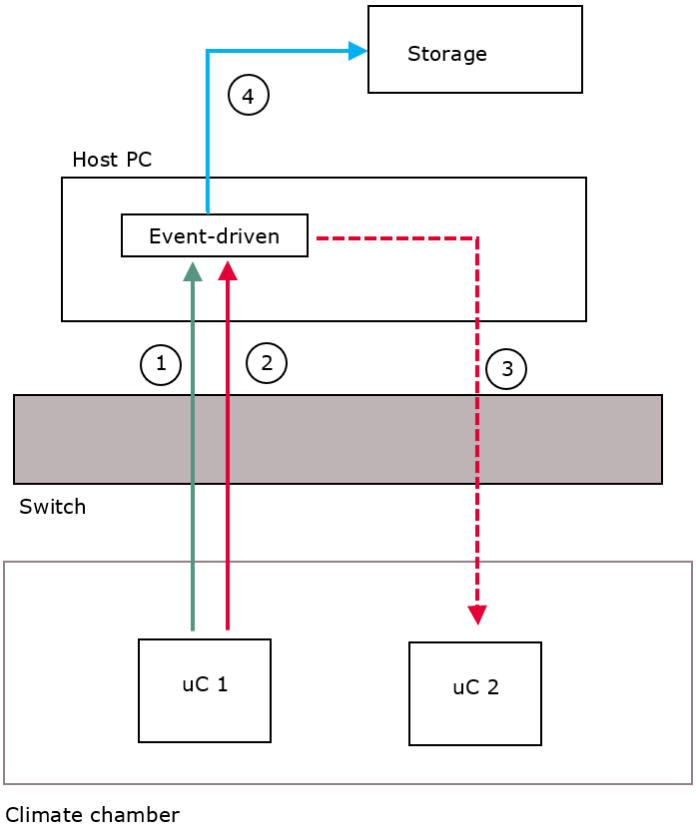


Figure 1.1: Existing architecture for distributed power stress test

1.1.1 Challenges and motivation

The distributed test setup architecture depicted in Figure 1.1 has two *control nodes* (*uC 1* and *uC 2*) which are usually connected to one DUT each. A central *Host Personal Computer (PC)* controls the operation of these *control nodes* through an Ethernet communication link (green/red arrows). These *control nodes* are responsible to perform different stress on their connected DUTs (e.g., power cycling of the attached DUT), collect periodic measurement data from the DUT (e.g., through Serial Peripheral Interface (SPI)) and communicating back the results to the central *Host PC*. Apart from collecting and transmitting the sensed data back to the *Host PC*, *control nodes* can also receive actuation signals from the central *Host PC* (based on the sensor inputs received) to adjust various stress parameters (like a closed control loop).

The control nodes and their corresponding DUTs will be inside an enclosed testing environment with specific temperature conditions (represented as *climate chamber*).

The Ethernet communication link between the central *Host PC* and the *control nodes* are mediated through an intermediate Ethernet *Switch*, so that it'll be easy to add another *control node* in future (scalable). The measurement data reaching *Host PC* is also logged to an extra storage space as represented in the Figure (blue arrow, (4)).

There is currently two classes of data collected and transmitted to the *Host PC*:

1. Critical measurement data that requires immediate action/actuation (e.g., for completing the control loop)

2. Non-critical measurement data that is continuously streamed back to the "Host PC" for debugging/logging purposes.

The non-critical data streaming from the control node (*uC 1*) to *Host PC* is represented by (1), the green arrow in Figure 1.1. The red arrowed straight line (2) from the control node *uC 1* to *Host PC* represents the critical measurement data. This periodically sent critical data will be processed on the *Host PC* to generate a specific event (*event-driven* logic), such that a control actuation/response to the event will be immediately transmitted to another *control node* (represented by the red dotted arrow (3) in the figure).

The main challenge associated with the "centralised event-driven logic" employed by the *Host PC* is that it requires a substantial amount of "response time". "response time" is the time taken to act from the sense of a critical measurement data (*uC 1* to *Host PC*) to the transmission of actual actuation data (*Host PC* to *uC 2*). The large latency or "response time" occurred is also because of the effect of interference caused on the 'critical data' transmission from the 'non-critical' data streaming.

Another challenge associated with the current data management and distribution (Ethernet data communication between *control nodes* and *Host PC*) scheme employed is that all the data communication is carried out through naked User Datagram Protocol (UDP)/Internet Protocol (IP) socket communication. To add another *control node* to the switch/network, it requires a substantial amount of code modification to be carried out in the *Host PC* software. Due to the lack of a non-industry standard (primitive) data management and distribution scheme, modern data visualisation/debugging tools cannot be easily integrated into the *Host PC*. This limitation forces the test engineers to stick to proprietary visualisation tools that are designed specifically for the current software stack.

1.2 Scope and objective

The main goal of the thesis is to give deterministic guarantees to the time-critical data transmission (e.g., control loop data) over Ethernet. This also means that the huge latency/"response time" observed in order to act upon measurement data from the central *Host PC* should also be reduced. Creating a scalable architecture that can seamlessly add several control nodes to the network shall also be considered while designing the "low-latency deterministic" communication. The choice of the scalable protocol should adhere to the industrial standards and should support easy integration of other visualisation tools.

Based on the challenges identified in the previous section, the following research questions are formalised to align with the goal of the thesis:

1.2.1 Research questions

1. What are the architectural modifications or additions that need to be done on the distributed test setup, so that the time critical control data can be transmitted (control loop between *control nodes*) deterministically over Ethernet, by reducing the current "response time"?

2. What is a suitable industrial standard communication protocol available that will seamlessly allow the addition of another *control node* (scalable) into the network (without much overhead), and which also supports easy integration of modern data visualisation tools?

The research questions are answered through the systematic methods explained in the next section.

1.3 Research methods

The important research methods used to answer the research questions were experimental study, literature review and data analysis. A literature review related to industrial communication protocol standards for M2M communication helped in answering the second research question. Online documentation, protocol specifications, already existing Internet of Things (IoT) implementation reviews comparing different

protocol standards etc. were the main sources of reviewed contents. Literature review on deterministic communication (AVB, TSN standards) and time synchronisation (Precision Time Protocol (PTP)) helped initially to select an appropriate solution to answer the first research question. These include research papers and journals related to the current IEEE industrial TSN and AVB communication standards, video sources and other online documentation. The data collected after actual implementation based on a real-time Operating System (OS) was further analysed to draw conclusions. Those implementation results justified the design choices and thus helped in answering the research questions.

Apart from the research methods constant brainstorming, review meetings and discussions with senior engineers and supervisors at KAI gave better insights into current architecture and exact product expectations to be met, in order to answer the research questions.

1.4 Thesis outline and structure

The rest of the thesis is organised as follows. Chapter 2 describes all the prerequisites and literature necessary to understand the rest of the thesis. Chapter 3 presents the proposed solution which will help answer the research questions. Chapter 4 covers the comprehensive description of the modules that were implemented on the Zephyr RTOS to support Infineon XMC4500 reference board and to enable us to establish Ethernet communication, time synchronisation etc. Chapter 5 explains the experimental setup that was implemented (both software and hardware) to answer the research questions. The comprehensive results based on the conducted experimental study and their analysis are presented in Chapter 6. These results explain the achieved deterministic behavior along with clear answers to the research questions. Chapter 7 draws further conclusions based on the analysed results and recommends future improvements to be done on the existing implementation.

2

Background

The basic concepts and theory that is needed to understand the concepts introduced in this thesis are discussed in this chapter. Networking terminologies, M2M communication technologies, literature reviews on deterministic Ethernet protocols and standards etc. are discussed. A brief introduction of Zephyr RTOS along with an overview of the Ethernet peripheral of the Infineon XMC4500 microcontroller is also included.

2.1 OSI and TCP/IP network models

The conceptual network model consisting of seven standard layers that computer systems use to communicate over a network is defined by the Open Systems Interconnect (OSI) model.

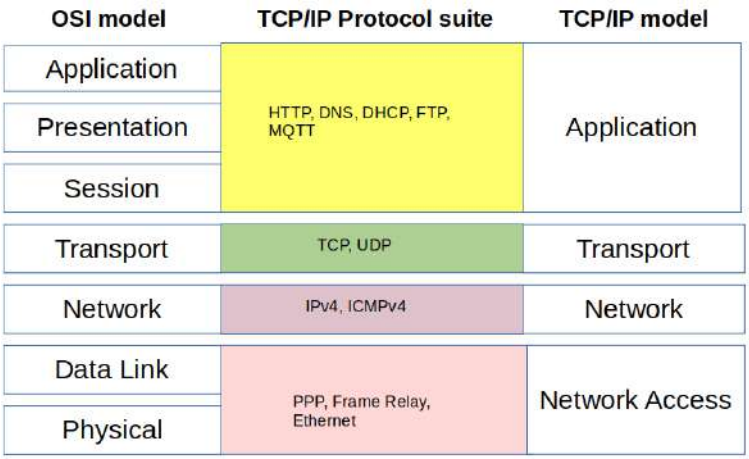


Figure 2.1: OSI and Transmission Control Protocol (TCP) network models.

Figure 2.1 represents the OSI model on the left side. Each intermediate layer serves a class of functionality to the layer above it and is served by the layer below it. The

TCP/IP model shown on the right side is the network model that is widely used for internet communication. TCP/IP implements its network functionalities into four layers by merging one or more layers from the OSI model. Implementation of standardised communication protocols in the software will realise the functionality of each layer (*TCP/IP Protocol suite* describes the implemented software protocols for the TCP/IP model in Figure 2.1).

Each layer contributes toward the network communication by encapsulation and decapsulation of data. When the sender needs to transmit data over a network, its encapsulated from top to bottom layer (from application to physical) by appending metadata/header specific to each layer. The final packet will be converted into serial bits and placed on the medium (eg. Ethernet cable) by the physical layer for transmission. Similarly, the received bits will be decapsulated from the bottom to the top layer, by removing the headers and extracting the data of interest.

2.1.1 TCP and UDP

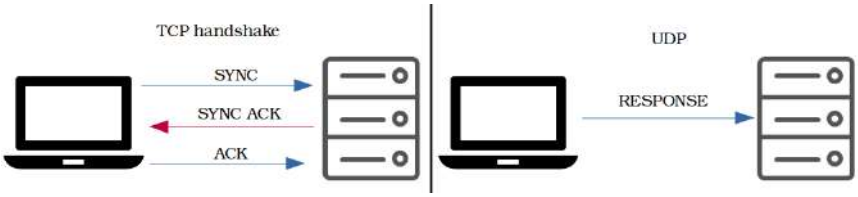


Figure 2.2: TCP and UDP communication pattern

The standard transport protocols placed above the IP layer for network communication are TCP and UDP. TCP is a connection-oriented protocol. Connection-orientation means that the communicating devices should establish a connection before transmitting data and should close the connection after transmitting the data. UDP is the Datagram-oriented protocol. This is because there is no overhead for opening a connection, maintaining a connection, and terminating a connection. UDP is efficient for broadcast and multicast types of network transmission. Figure 2.2 represents the exchange of Synchronization (SYNC)/Acknowledge (ACK) messages that is needed for maintaining a TCP connection. On the other hand, UDP requires very minimal bandwidth and fewer message exchanges for data transfer.

The heavy traffic introduced by the connection-oriented TCP transport makes it much slower than UDP. On the other hand, TCP also provides a more reliable (reliability through retransmission of lost packets and guaranteed delivery of data through congestion-free dedicated path) and error-free communication compared to

UDP. TCP has a 20 B to 60 B variable length header while UDP has an 8 B fixed-length header.

2.1.2 Unicast, Multicast and Broadcast

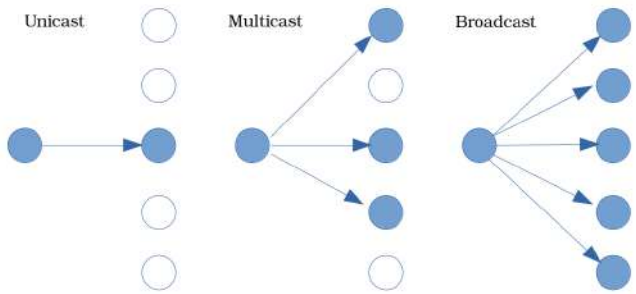


Figure 2.3: Unicast, Multicast and Broadcast

Network communication in which one source device sends data to a destination device on the network is known as unicast communication. The destination IP address of the destination device will be unique to that device in the network. When the source device sends data to multiple devices, but not all devices on the network, the communication is known as multicast. Special class D multicast IP addresses are used for grouping together the destination devices. When the source needs to send to all the devices in the same network, the communication model is broadcast and a special IP address is reserved as the broadcast address. Figure 2.3 represents the unicast, multicast and broadcast communication pattern.

2.2 Ethernet

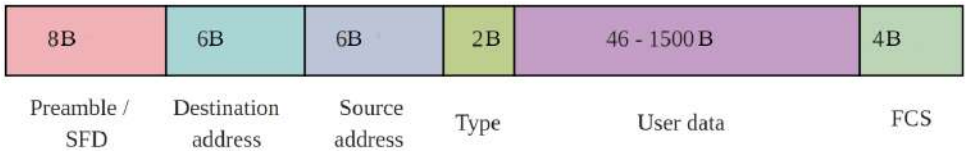


Figure 2.4: Ethernet frame

Ethernet is the popular wired communication protocol standard that's used in networking infrastructures like local area network (LAN), Metropolitan Area Network (MAN) and Wide Area Network (WAN). Ethernet was commercially available in 1980 and was standardised in 1983 as IEEE 802.3 [2]. The initial versions of the Ethernet

used shared, coaxial cables and it was replaced by twisted pair cables for supporting full-duplex communications. This leads to the development of 10BASE-T, 100BASE-T and 1000BASE-T, which support a bandwidth of 10, 100 and 1000 Mbit/s.

Ethernet operated across layer 1 and layer 2: Physical layer and Data link layer. IEEE 802.3 aims to address the lower portion of layer 2 - Media Access Control (MAC) and layer 1. Figure 2.4 shows the composition of the Ethernet II frame. The basic frame consists of seven elements split between three main areas:

1. **Header:** *Preamble/Start of Frame Delimiter (SFD)*: The preamble consists of 7 bytes long altering ones and zeros pattern. It's used for indicating the start of a frame and for synchronisation purposes. SFD is the one-byte altering ones and zeros pattern ending with two ones. The Ethernet frame follows the end of SFD.

   Destination address: The receiver MAC address to which the Ethernet frame must be sent.

   Source address: The sender MAC address from where the Ethernet frame is transmitted.

   Length/Type: The 2-byte field specifies the length of the entire Ethernet frame and can represent values from 0 to 65534.

2. **Payload:** *User data*: This block contains the payload data and it may be up to 1500 bytes long. If the length of the field is less than 46 bytes, then padding data is added to bring its length up to the required minimum of 46 bytes.

3. **Trailer:** *Frame Check Sequence (FCS)*:The 4-byte long FCS contains Cyclic Redundancy Check (CRC) sequence for error checking. It's calculated based on the *Source Address*, *Destination Address*, *Length/type* and *Data* fields.

Every Ethernet Network Interface Card (NIC) is given a unique identifier called a MAC address. This is assigned by the manufacturer of the card and the MAC address for Ethernet is a 48 bit (6 B) number which must be unique on the network.

2.3 M2M communication

M2M communication enables devices of the same capability to communicate among themselves using a unified communication methodology by reducing the network overhead. M2M combines information technology with machine data communication between devices or machines, enabling applications like industrial instrumentation. The popular M2M communication protocols that are existing in the IoT and *Industry 4.0* domain are discussed below.

2.3.1 MQTT

MQTT [3] is a lightweight application layer protocol that follows a topic-based Publish-Subscribe communication pattern. MQTT is a relatively simple M2M protocol designed specifically for constraint devices with low resources like memory, processor power and battery. Because of its requirements for ordered, lossless, bi-directional connections, MQTT uses TCP/IP as its transport layer. Even though MQTT uses TCP/IP, the small transport overhead (fixed length of 2 bytes) makes the MQTT an interesting solution for unreliable networks with restricted resources, such as low bandwidth and high latency.

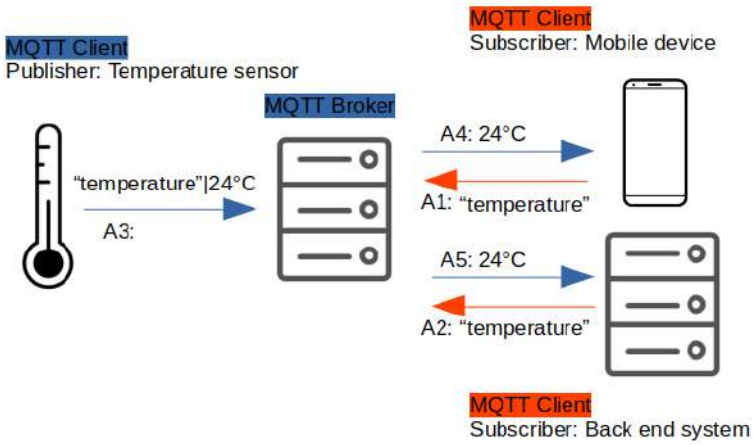


Figure 2.5: MQTT Publish/Subscribe architecture

MQTT specifies two different network entities named *message broker* and *clients*. *Message broker* (shown as *MQTT broker* in Figure 2.5) is the central entity that is responsible to manage client connections and routing the MQTT publish messages to corresponding subscribed clients. MQTT client is the device (publisher or subscriber) which will be connected to the *broker* through its MQTT client library over the network.

Information is organized in a hierarchy of topics. When a publisher has a new item of data to distribute, it sends a control message with the data to the connected broker. The broker then distributes the information to any clients that have subscribed to that topic. The publisher does not need to have any data on the number or locations of subscribers, and subscribers, in turn, do not have to be configured with any data about the publishers. Thus with the MQTT broker architecture, the client devices and server applications become decoupled.

Figure 2.5 shows an example Publish/Subscribe architecture of MQTT. The MQTT clients acting as a subscriber (mobile device and back-end system) initially connects with the *MQTT Broker* and sends out subscription messages for the topic *temperature* (represented by A1 and A2). Similarly, the MQTT client acting as a publisher (Temperature sensor) connects with the broker. The temperature readings will be then published as payloads of the MQTT PUBLISH message with the topic *temperature* (represented by A3). The MQTT broker will route these temperature readings to the subscribed clients (represented by A4 and A5).

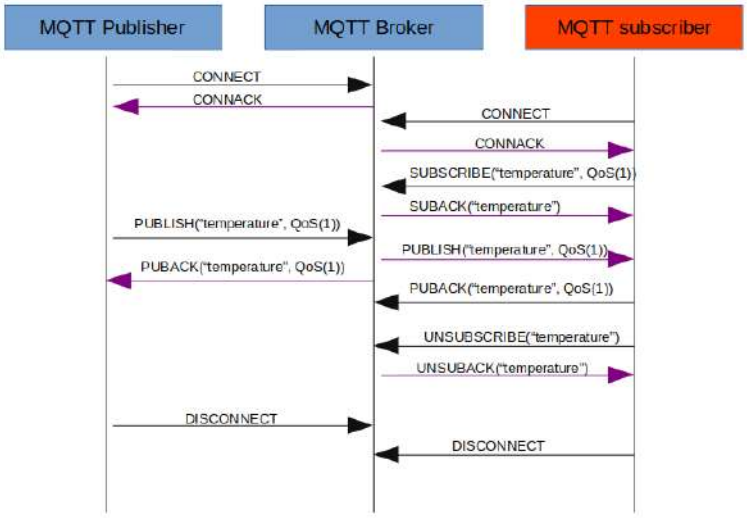


Figure 2.6: MQTT communication sequence diagram

Figure 2.6 represents the communication sequence diagram of the same MQTT Publish/Subscribe example described above with QoS 1. MQTT provides another layer of reliability using three levels of QoS named QoS 0, QoS 1 and QoS 2.

1. *QoS 0: At most once (fire and forget)*: A simple request is sent without expecting a response, leaving the assurance of delivery to TCP.

2. *QoS 1: At least once*: At QoS level 1, the message is followed by a similar ACK response. In case of missing ACK, the message will be sent again.

3. *QoS 2: Exactly once*: At QoS level 2, a four-way handshake is to be established to ensure the message delivery only once.

There are a total of 16 types of messages defined by the MQTT specification for broker-client communication and acknowledgments. The popular open-source implementation of the MQTT broker is *mosquitto* [4] by the Eclipse foundation.

2.3.2 MQTT-SN

MQTT-SN [5] stands for "MQTT for Sensor Networks" which is aimed at embedded devices on non-TCP/IP networks (such as Zigbee). Its official site says: *MQTT-SN is a publish/subscribe messaging protocol for wireless sensor networks (WSN), with the aim of extending the MQTT protocol beyond the reach of TCP/IP infrastructure for Sensor and Actuator solutions .*

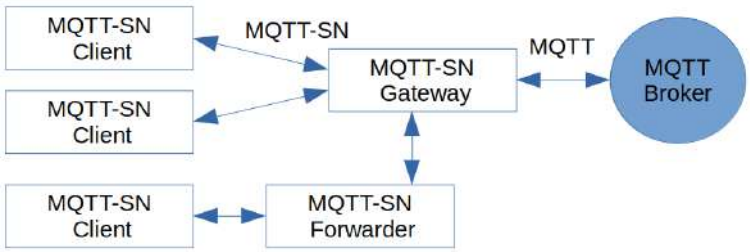


Figure 2.7: MQTT-SN architecture.

Figure 2.7 represents the MQTT-SN architecture. The main three components are MQTT-SN client, MQTT-SN gateway and MQTT-SN forwarder. MQTT-SN client connects with the MQTT broker via the gateway. MQTT-SN gateway translates the MQTT-SN messages to MQTT message and vise versa. If the MQTT-SN client is not directly connected to the gateway, the MQTT-SN forwarder can be used to forward the MQTT-SN messages to the gateway *unchanged*.

MQTT-SN was targeted for low-power battery-operated sensors with very limited processing power and storage. These constrained devices usually have limited payload size requirements and will not be always on (frequent sleeping).

The main differences between MQTT and MQTT-SN are:

1. Reducing the size of the message payload

2. Removing the need for a permanent connection by using UDP as the transport protocol.

MQTT-SN also introduced *topic-ids*, *short topic names* and *pre-defined topics* for reducing the message size during publishing and simplifying the transmit process. MQTT-SN supports three levels of QoS from -1, 0 and 1 (highest QoS). There is no need for any connection setup, tear down, subscription or registration in case of clients publishing with QoS -1 (fire and forget). The MQTT-SN specification [5] says: *The*

client just sends its PUBLISH messages to a GW (whose address is known a-priori by the client) and forgets them. It does not care whether the GW address is correct, whether the GW is alive, or whether the messages arrive at the GW.

Eclipse Paho [6] is a popular open source MQTT-SN implementation.

2.3.3 OPC UA

Open Platform Communications United Architecture (OPC UA) [7] is a data exchange standard for industrial communication (M2M and PC-to-Machine). OPC UA is a platform-independent service-oriented architecture that integrates all the functionality of the individual OPC classic specifications into one extensible framework.

Although OPC classic was built on Microsoft Windows technology using the COM/D-COM (Distributed Component Object Model) for the exchange of data between software components, the OPC UA supports advanced features like *Service Oriented Architecture (SOA)*, *Publish-Subscribe (PubSub)*, etc.

2.3.4 CoAP

One of the main communication paradigms is the *Request/Response* model. It's commonly used in the distributed system to exchange information through messages between receiver and sender. The usual case where a client sends out a request to the server and the server in turn services the request through a response is the typical flow. Constrained Application Protocol (CoAP) follows the *Request/Response* model.

The official site of CoAP [8] says: *"The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained networks in the Internet of Things. The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation."*

This protocol is described in RFC 7252 [8] and is taken forward by IETF Constrained RESTful Environments (CoRE) working group.

Based on the REST model like Hypertext Transfer Protocol (HTTP), in CoAP also the servers make resources available under a uniform resource locator (URL), and clients can access these resources using methods such as GET, PUT, POST, and DELETE. The close resemblance of CoAP with HTTP makes it very user-friendly and also makes it easier for making them connected easily using application-agnostic cross-protocol proxies. Like HTTP, it can also carry different types of payloads and is able to integrate with XML, JSON, CBOR, or any data format of choice. CoAP also supports strong security capabilities in its offerings.

2.3.5 OMA LWM2M

The M2M or IoT device management and service enablement protocol from Open Mobile Alliance (OMA) is Light Weight M2M (LWM2M) [9]. LWM2M communication protocol defines the communication interface between the LWM2M server and the LWM2M client. The LWM2M clients residing in the IoT devices will enable the devices to be managed in the IoT ecosystem, supporting the features like remote provisioning of security credentials, firmware updates, connectivity management (e.g. for cellular and WiFi), remote device diagnostics and troubleshooting.

LWM2M protocol enables devices from multiple vendors to co-exist and the main supported features for service enablement are sensor and meter readings, remote actuation and configuration of host devices. Initially, LWM2M was built on CoAP and later implemented support for additional transport layers.

2.4 Deterministic Ethernet Protocols for Real-time Systems

2.4.1 ARINC664

The Ethernet-based network for avionics system is defined by the ARINC664 [10] standard (also known as *Avionics Full-Duplex Switched Ethernet*). It provides reliable and deterministic communication by allowing designers to calculate the maximum latency of every message delivered. ARINC664 does not use global time synchronisation and all traffic is considered as a single class called *guaranteed service* with bounded end-to-end delay. This standard borrowed concepts like *token bucket* and *asynchronous transfer mode* to reserve dedicated bandwidth for mission-critical applications for providing deterministic QoS [10].

2.4.2 Time-Triggered Ethernet

The Time-Triggered Ethernet [11] (SAE AS6802) known as TTEthernet or TTE provides time-critical deterministic Ethernet communication by implementing fault-tolerant time synchronisation and synchronous packet switching. Time-scheduling is employed on the synchronised nodes to deliver deterministic real-time traffic, and also allowing mixed-critical traffic to co-exist. Layer 2 of the OSI model is implemented by the TTEthernet and is compatible with IEEE 802.3 standard. The main traffic classes and message types provided by TTE [11] are:

1. **Synchronization Traffic (Protocol Control Frames - PCF):** The highest priority PCF frames and their interfaces are used to establish and maintain high precision fault-tolerant clock synchronisation.

2. **Time-triggered traffic:** After establishing synchronised local clocks "Time-triggered" Ethernet packets are scheduled to be sent over the network at predefined time to achieve deterministic communication with precise guaranteed delay.

3. **Rate-constrained traffic:** Applications with less stringent real-time requirements will use this traffic to define the maximum upper bound of latency and jitter. The bandwidth is predefined according to the upper bound.

4. **Best-effort traffic (incl. Virtual-LAN (VLAN) traffic):** The rest of the low-priority traffic is sent with this class. Guarantees about the arrival time, delay, or jitter are not provided in this case.

2.4.3 AVB and TSN standards

AVB is the standard that was aimed at reducing latency and increasing the reliability of audio/video streams in switched Ethernet networks. Development of AVB was initially done by the IEEE Audio Video Bridging task group of the IEEE 802.1 standards committee and later renamed as TSN task group. The relevant standards related to the thesis are discussed below.

2.4.3.1 IEEE 802.1AS

Deterministic latency is achieved through time synchronisation and the application of a global schedule [12]. The global schedule can be viewed as driving through busy city streets without having to slow down or stop at any cross junctions. This is possible only if other cars in the streets know *when* to cross and *when* to stop, when *critical traffic* is passing through. Hence a global reference time should be maintained by all the nodes in the network to follow a single global schedule deterministically. The Timing and Synchronization standard defined by IEEE 802.1AS [13] uses the modified version of IEEE 1588 [14] (PTP), called generic Precision Time Protocol (gPTP) for time synchronisation.

gPTP first selects a grand master clock from the available candidates through the Best Master Clock Algorithm (BMCA) and assign it as the global reference clock/time.

The grand master will periodically propagates its timing information through hardware timestamped PTP packets to the other nodes (slaves). Offset and delay correction calculation is done by the slave after extracting the timing information from the packets to be synchronised with the master. The periodic propagation and correction of timing information keep all the node's time synchronised (up to 10ns accuracy can be achieved with gPTP).

2.4.3.2 IEEE 802.1Qat

The Stream Reservation protocol [15] (IEEE 802.1Qat) guarantees end-to-end bandwidth for the network flow, so that packets being dropped due to congestion is avoided. It specifies admission control criteria for the flow of data (streams) based on the application resource requirements and the available network resources, to ensure QoS requirement of the entire flow is satisfied during the entire path.

2.4.3.3 IEEE 802.1Qav

IEEE 802.1Qav - Forwarding and Queuing Enhancements for Time-Sensitive Streams [16], is known as Credit Based Shaper (CBS). Each stream admitted through the reservation protocol (as mentioned in Section 2.4.3.2) is assigned priority values (to classify as class A and class B traffic) and further enqueued into different queues.

A *token bucket* [17] based algorithm is used to send out the packets from the queue. Tokens are added to the bucket at a constant rate (*idleSlope*) while the transmission is paused/blocked and the overall credit for that particular queue accumulates. Packets can be dequeued and transmitted when there's a positive credit, and tokens will be taken away during transmission (*sendSlope*). Packets will be queued if there are not enough tokens. This results in spreading the transmission of packets across the time domain and will reduce the sudden bursts of packets, hence decreasing bridge/forwarder buffer size and congestion. It'll also prevent low priority traffic from starving due to the excessive transmission of high priority traffic.

2.4.3.4 IEEE 802.1Qbv

The *Time Aware Shaper* defined by the IEEE 802.1Qbv [18] standard uses the basic principle of Time Division Multiple Access (TDMA) for scheduling traffic and to give

real-time guarantees [19]. The entire Ethernet communication (with mixed priority traffics) is divided into periodic cycles or schedules and repeatedly transmitted. Each cycle is again logically divided into time slices that are placeholders for specific traffic (with certain priority encoded into VLAN tag), giving them exclusive access to the medium while transmitting. This exclusive time sliced transmision will enable time-critical traffic to be temporarly isolated from the background non-critical traffic, enabling deterministic trasmission guarantees.

A worst-case latency of 100 µs over 5 hops and maximum transmission period of 500 µs is provided by the IEEE 802.1Qbv.

2.4.3.5 IEEE 802.1Q priority mapping and strict transmission selection

		Available traffic classes							
		1	2	3	4	5	6	7	8
Priority	0 (Default)	0	0	0	0	0	1	1	1
	1	0	0	0	0	0	0	0	0
	2	0	0	0	1	1	2	2	2
	3	0	0	0	1	1	2	3	3
	4	0	1	1	2	2	3	4	4
	5	0	1	1	2	2	3	4	5
	6	0	1	2	3	3	4	5	6
	7	0	1	2	3	4	5	6	7

Figure 2.8: IEEE 802.1Q traffic class mapping

IEEE 802.1Q [20] was introduced to provide determinism to the standard Ethernet (IEEE 802.3) among other things. A 4-byte special IEEE 802.1Q header was included in the traditional 802.3 Ethernet frame called VLAN tag, which enabled the users to assign priorities to the Ethernet frame. The 3-bit PCP (Priority Code Point) present in the VLAN tag can be assigned with one out of eight priority levels (as shown in eight rows of Figure 2.8, second column).

In order to implement different levels of QoS between different types of Ethernet frames, *traffic class mapping* is used. *Traffic class mapping* is a process in which single or multiple PCP values are grouped into a different traffic class called CoS (Class of Service). Figure 2.8 gives the recommended *traffic class mapping* by IEEE according to the QoS requirements (i.e. based on the number of traffic classes required for the application).

Each CoS is associated with a buffer to hold its Ethernet frames. For instance, a traffic class mapping of 2, will be having two queues where PCP values from 0 to 3 will be classified into a first queue and others into the second queue. Ethernet frames to send will be enqueued into an appropriate traffic class queue based on their PCP values and the *traffic class mapping* (Figure 2.8). Later scheduling policies like *strict transmission selection* is employed on the queues to transmit packets, ensuring required QoS or determinism. Packets from the higher CoS queue will be sent completely first, followed by the lower priority CoS queues. Minimum latency is guaranteed on packets belonging to higher CoS under congestion based on the *strict transmission selection*.

2.5 Zephyr RTOS

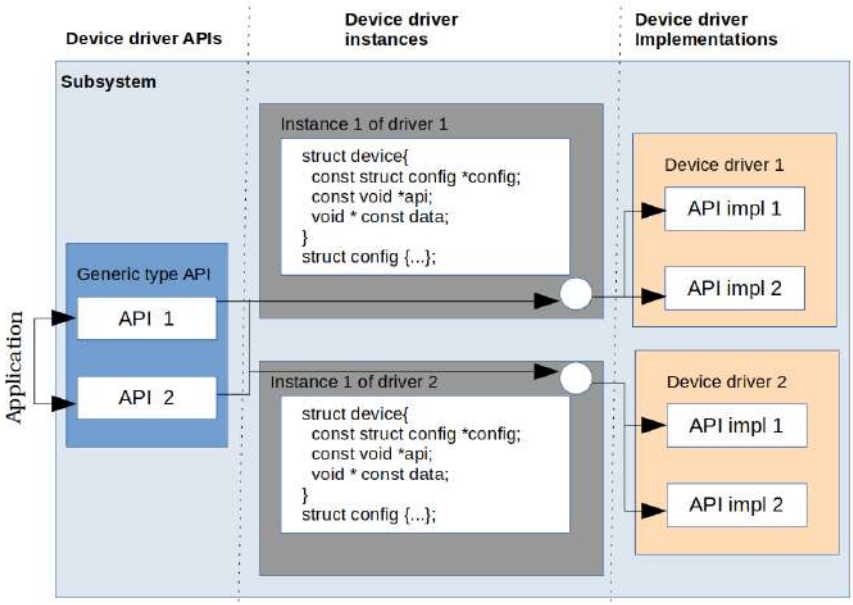


Figure 2.9: Zephyr device driver model

Zephyr [21] RTOS is the small-footprint open-source real-time kernel designed for resource constraint embedded devices (e.g. for IoT applications), which support wide variety of instruction set architectures like ARMv7. Advanced power management services, inter-thread synchronisation services like semaphores, compile-time registration of interrupt handlers and implementation of preemptive and non-preemptive schedulers, implementation of important communication protocol stacks with support for *POSIX* application programming interface (API)s etc are the main features of Zephyr. Another feature is the *compile-time resource* definition by which code size is decreased and performance is increased for resource constraint devices.

The optimised device driver model of Zephyr is depicted in Figure 2.9 which provides a consistent way to configure different vendor-specific driver implementations and also to initialise them. Application use only the generic sub-system APIs exposed by Zephyr is device-independent. Each vendor will implement the generic (device-independent) sub-system driver API structure according to the device's capabilities. Additional device-dependent features can also be added to the driver.

Figure 2.9 shows an example where the applications use the generic APIs *API 1* and *API 2* of the subsystem. Different vendors implement those generic APIs based on their devices as *Device driver 1* and *Device driver 2*. Each driver implementation can have more than one instance based on the requirements and different *struct device* will be initialised for each instance. Zephyr device driver uses *devicetree* [22] to describe the hardware and each device-specific implementation is assigned to the generic device API function pointers (*const void *api*).

2.6 Infineon XMC4500 Ethernet

Infineon XMC4500 [23] microcontrollers belong to the XMC4000 family of high-performance, energy-efficient microcontrollers optimised for industrial applications. This ARM Cortex-M4-based microcontroller is equipped with a 16 KB on-chip boot Read-Only Memory (ROM), two 64 KB on-chip instruction and data memory, 32 KB on-chip high-speed communication memory etc. XMC4500 also supports Ethernet MAC IP capable of 10/100 Mbit/s transfer rates with Direct Memory Access (DMA) support. Universal Serial Bus (USB) 2.0 and Full-speed OTG, Controller Area Network (CAN) , Universal Asynchronous Receiver Transmitter (UART), SPI etc are some other communication peripherals supported by XMC4500.

A detailed explanation of XMC4500 Ethernet MAC IP is given in *Chapter 15* of the XCM4500 Reference manual [23]. *ETH Core*, *ETH DMA*, *ETH MTL (MAC Transaction Layer)*, *System time module* and *MAC Management counters* are the main five functional units of the MAC IP.

The Ethernet frames from the user application are first enqueued into a 2K First In First Out (FIFO) buffer memory which is provided by the *ETH MTL*. The *ETH DMA* can be made use by the application to offload writing/reading of frames to/from the FIFO. The *ETH Core* is responsible for transmitting the frames from the FIFO to the Ethernet Physical Layer (PHY) via the Media Independent Interface (MII)/Reduced Media Independent Interface (RMII) interface. Advanced timestamping of transmit/receive frames is done by the *System time module*. A detailed report about the bus statistics of Ethernet frames will be carried out by the *MAC Management counters*.

2.6.1 PTP System time update

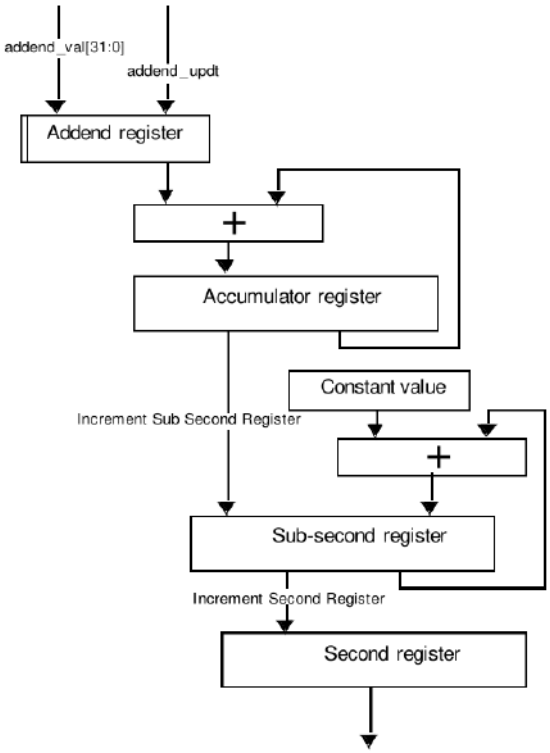


Figure 2.10: System time update using fine update

System time module maintains a 64-bit counter (ETH0_SYSTEM_TIME_SECONDS and ETH0_SYSTEM_TIME_NANOSECONDS), which is used as the time source for taking Ethernet frame timestamps. Figure 2.10 represents the register structure and logic used for *fine update* method by which the system time counter is updated. The gradual update of the system time for synchronising master clock with slave clock with minimum jitter (without drastic changes) is defined by the *fine update*.

As shown in Figure 2.10 the 32-bit *Accumulator register* repeatedly sums up the 32-bit *Addend register* to generate a carry pulse. This arithmetic carry will increment part of the system time counter (*Sub-second register*), making the *Accumulator register* a high precision multiplier or divider [23]. Later the arithmetic carry from the *Sub-second register* will increment the *Second register*. In order to achieve a 20 ns accuracy, a decimal value of 43 is used to accumulate the *Sub-second register* in the presence of 50 MHz clock frequency. Detailed PTP driver implementation details on Zephyr RTOS to achieve network time synchronisation is described in Section 4.2.1.

3

Proposed methods

This chapter outlines the methods that are proposed to address the research questions identified in Chapter 1. These proposals are formulated based on a careful analysis of the current architecture and the specific challenges associated with the test setup. The extensive literature review (previous work) related to deterministic Ethernet communication like TSN, AVB, and M2M protocols, along with frequent brainstorming with senior engineers helped to formalise the proposals.

3.1 Architectural modification

Figure 3.1 depicts the proposed distributed architecture to address the challenges of the current system. A detailed explanation of the proposal is given in the following subsections.

3.1.1 Overall latency/*response time* reduction

The first research question (Section 1.2.1) identified was related to (i) decreasing the overall latency (*response time*) of the time critical data communication and (ii) giving deterministic guarantees for the same data.

The overall latency identified was largely due to the *centralised event-driven logic* residing in the *Host PC* (Figure 1.1). The time critical sensor data collected by *uC 1* needed to be first transmitted to the *Host PC* and then relayed back to *uC 2* (the decision to send the actuation command to *uC 2* is based on an *event-trigger* logic in *Host PC*). The presence of the intermediate *Host PC* between the communication path of *uC 1* and *uC 2* adds transmission delay. The extra latency added due to the processing time taken for the *event-driven logic* will also contribute to the increased delay.

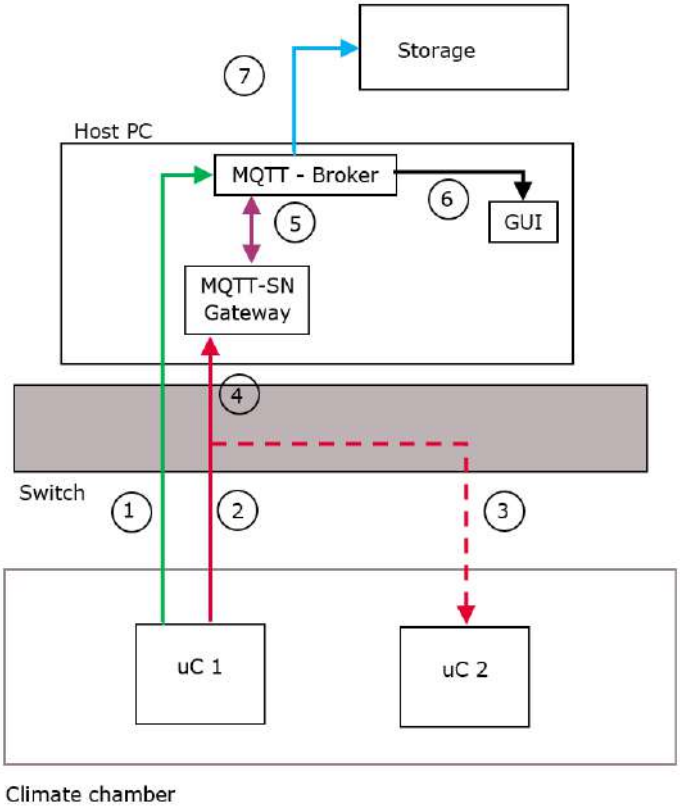


Figure 3.1: Proposed scalable architecture for distributed power stress testing

The overall latency due to this centralised architecture could be reduced by decentralising the critical data communication path between *uC 1* and *uC 2*. Thus a direct source-to-destination communication between sensor-actuation nodes can be employed to achieve decentralisation. The multicasting feature of IPv4 networking (Section 2.1.2) can be utilised to send one critical sensor data to multiple destinations directly through the intermediate Ethernet switch. The red arrows (normal and dotted) with numbers (2) and (3) in Figure 3.1 depicts this multicasted traffic from *uC 1* to *uC 2*.

3.1.2 Scalability

The second research question is to improve the scalability of the *control nodes* in the distributed setup. The simple lightweight TCP/ IP based messaging protocol called MQTT is proposed for the scalability improvement. MQTT has become one of the popular choices for transmitting data between machines (M2M) through a simple

topic based Publisher/Subscriber mechanism (as detailed in Chapter 2). MQTT was originally designed specifically for IoT devices with low bandwidth and power, for measurement and monitoring use cases. MQTT is widley used by *Industry 4.0*, IoT and other industrial domains, and it is also approved by The Organization for the Advancement of Structured Information Standards (OASIS). This *ISO/IEC 20922* standard with advanced features like security (e.g. Transport Layer Security (TLS) based encryption and Open Authorisation (OAuth) based client authentication), built-in scalability, multiple QoS levels, simple and lightweight implementations etc. makes MQTT one of the best candidates for the current application.

The non-critical measurement data can be streamed directly from the *control node* to the MQTT broker residing in the *Host PC* (green arrow from *uC 1* labelled as (1) in Figure 3.1). The *control node* which is a MQTT client, acts as a *Publisher* in this case. The *control nodes* will be publish the measurement data as a payload of a specific MQTT topic. For example, temperature data will be published with the topic "temperature/control_node_1". This makes it easy for the standard MQTT based visualisation tools (black arrow towards *graphical user interface (GUI)* labelled with (6)) to subscribe to topic like "temperature/* " to receive all temperature updates from all the nodes.

Adding a new *control node* to the network is also simple, as the node can directly open a TCP/IP based MQTT session with the broker and straightaway publish its temperature updates. For example, the new node can use the topic "temperature/-control_node_3" to publish. The visualisation tool is guaranteed to get the updates from the new node as the complete data management (MQTT topic based *Publisher* and *Subscriber* management) is handled by the MQTT broker. This creates a scalable plug-and-play data transmission system without any overhead (code change) and easy integration with standard industrial visualisation tools (e.g. OXYGEN™ measurement tool from Dewetron [24]).

The critical measurement data from the *control nodes* which will be multicasted (as explained in Section 3.1.1) to other *control nodes* should be also transferred to the *Host PC*. The UDP/IP based MQTT-SN protocol, specifically designed for embedded Sensor/Actuator solutions can be used for this use case. The data transmitted with UDP transport layer requires less bandwidth compared to the high bandwidth requirement of the TCP layer with multiple handshake messages and large packet size. This makes MQTT-SN suitable for the transmission of time sensitive critical data.

The MQTT-SN gateway residing in the *Host PC* will receive the incoming MQTT-SN publish traffic (the red arrow (4) from *uC 1*) and convert the message into

TCP based MQTT traffic to the MQTT broker(the purple arrow (5)) as shown in Figure 3.1. This ensures all the critical data is also delivered to MQTT *Subscriber* based visualisation tools by the MQTT broker. The *fire-and-forget* (QoS = -1) quality of service provided by the MQTT-SN protocol will be specifically suitable for the current multicast scenario because it requires no active involvement of the gateway or broker to publish a message to the subscriber. Thus the *uC 1* can directly send MQTT-SN publish message to the subscribing node *uC 2* without any additional latency. The gateway can act as a passive listener responsible for receiving all the publish messages to be relied back to the MQTT broker. The passive listening can be enabled by joining in the same multicast group.

The performance comparison of multicasted *fire-and-forget* (MQTT-SN with QoS-1) transmission against other MQTT/MQTT-SN transmissions (with different QoS levels) should be investigated to select the most deterministic medium for critical data transport. *Most deterministic* medium will be the one with less jitter and latency.

3.2 Deterministic Ethernet communication for the *critical measurement data*

While the proposed decentralised architecture (mentioned in Section 3.1.1) aims to reduce the overall latency/*response time*, this section describes the method that can be adopted to give deterministic guarantees to the *multicasted* critical data.

3.2.1 Zephyr RTOS

The current architecture has *control nodes* (Infineon XMC4500 microcontroller) running in-house firmware with limited functionalities. These limiting factors include a lack of standardised IEEE time synchronisation stack (IEEE 802.1AS), MQTT stack, advanced multithread scheduling etc. Zephyr® [21], an open source and lightweight RTOS developed by the Linux Foundation is selected to overcome the limitations imposed by the current firmware. The recent gPTP time synchronisation network stack implementations done by Intel and Antmicro [25] was also one of the main reasons for selecting Zephyr to implement deterministic Ethernet communication.

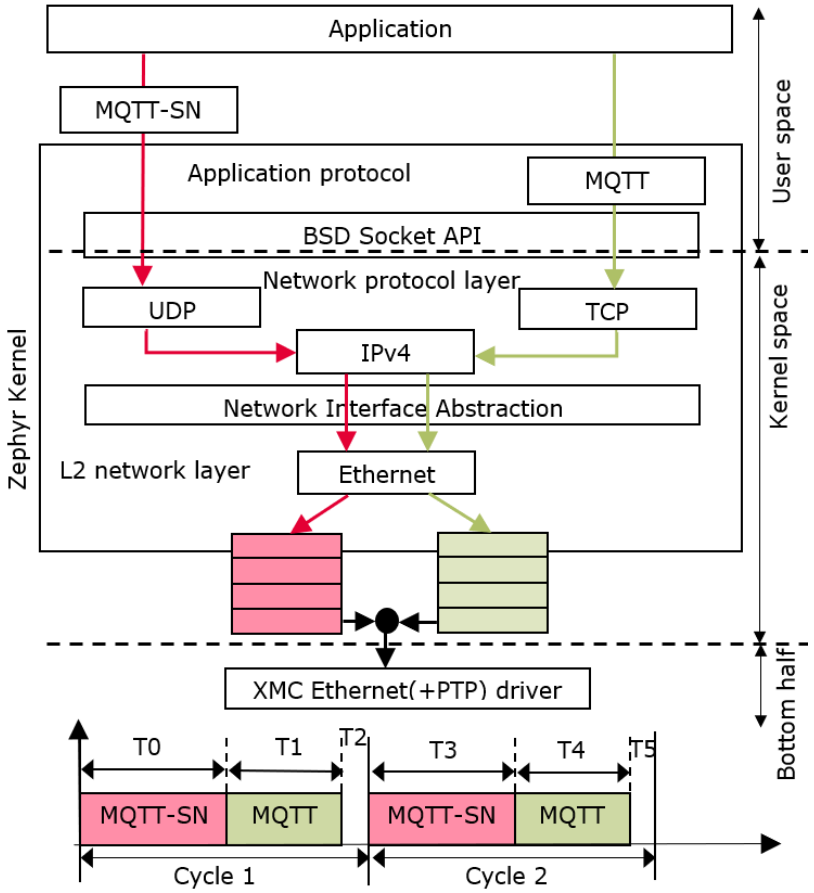


Figure 3.2: Proposed Ethernet data flow of critical/MQTT-SN (red) and non-critical/MQTT (green) communication from application layer down to the Ethernet device driver. Generic network schedule to minimise jitter is shown at the bottom.

3.2.2 Providing deterministic guarantees by reducing jitter

Both critical and non-critical data from each *control node* need to be communicated to the *Host PC* via an Ethernet link. Zephyr currently supports only basic features like SYSTICK (System Clock), Nested Vector Interrupt Controller (NVIC) and UART of Infineon XMC45-RELAX-KIT. To enable Ethernet communication, the complete Infineon Ethernet driver for Zephyr needs to be implemented in the kernel bottom-half as shown in Figure 3.2.

The control applications responsible for collecting measurement data or receiving actuation data are responsible for encapsulating and transmitting the corresponding

payloads using application layer messaging protocols (MQTT/MQTT-SN). Native MQTT protocol client APIs are implemented in the Zephyr RTOS for the application programmers to use. Open source MQTT-SN client implementation libraries providing MQTT-SN specific serialisation, deserialisation and transport layer like *Eclipse Paho* should be integrated into the Zephyr build system for transmitting critical data, as shown in the *User space* of Figure 3.2.

The control applications use the standard Berkeley Software Distribution (BSD) socket APIs exposed by the Zephyr kernel for network communication. The MQTT/ MQTT-SN stack will make use of the opened sockets for establishing their connection with MQTT broker/MQTT-SN gateway. The *MQTT* encapsulated measurement payload will be passed through the lower transport layers like UDP/TCP in the *Network protocol layer* of Zephyr. All the packets then obtain their IP packet metadata from the *IP stack* and are handed over to the *Layer 2 (L2) network layer*. Networking link layer operations and device-driver implementations are abstracted from the *IP stack* through the *Network Interface Abstraction* by the *L2 network layer*. The L2 layer for Ethernet will interact with the XMC Ethernet driver (implemented as per Zephyr device driver model) for transmitting Ethernet packets. The data flow of critical MQTT-SN data (represented by red arrow) and non-critical MQTT data (green arrow) through the Zephyr network stack is represented in Figure 3.2.

Due to the transmission of critical and non-critical traffic over one Ethernet peripheral, there's bound to be performance degradation of the critical traffic due to the presence of non-critical traffic. This is due to the fact that a critical packet will have to wait for all the non-critical packets, those that arrived first, to finish their transmission. Performance degradation can be measured in terms of the jitter experienced by the critical communication, which is the spread of latency in the arrival time of Ethernet packets. The logical solution to minimise the jitter effect is by separating both critical and non-critical data into different transmission queues and giving higher priority for the critical transmit queue. Thus creating a static schedule like transmission pattern (as shown in the bottom of Figure 3.2) of critical MQTT-SN and non-critical MQTT is desirable. Both MQTT and MQTT-SN transmit packets can be logically separated in the software by making use of different Zephyr network transmit queues.

Zephyr treats all network traffic, which are enqueued into a single queue, with equal priority by default. Zephyr config option can be used to set the number of transmit queues. Assigning priority directly from the BSD socket (using socket option SO_PRIORITY) for each traffic class will enable the Zephyr to assign packets to different transmit queues according to their assigned priority. Each traffic class queue corresponds to a specific Zephyr kernel work queue with a certain priority. This priority

mapping in Zephyr is according to the IEEE 802.1Q specification (as described in Section 2.4.3.5). Hence the desired deterministic behaviour can be ensured by utilising this traffic priority mapping from Zephyr RTOS.

4

Implementation details

The main implementation to enable deterministic Ethernet communication in XMC using Zephyr RTOS was the Ethernet driver module for XMC4500. The implementation details of the Ethernet driver along with the PTP driver (for network time synchronisation) and the integration of the *Paho* MQTT-SN library in Zephyr RTOS is explained in this chapter.

4.1 Ethernet communication

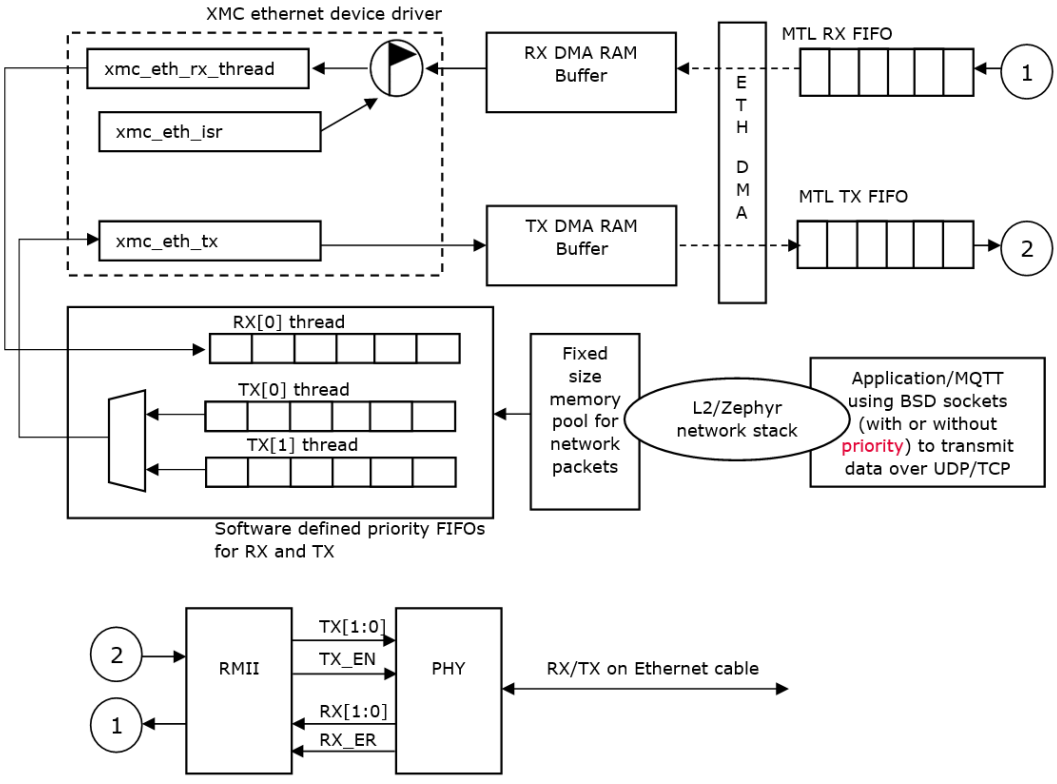


Figure 4.1: Ethernet communication data flow

The complete data flow path of an Ethernet packet from the application space to the physical Ethernet cable (and vice versa) is represented in Figure 4.1. Each data path corresponding to sending/receiving direction is explained below:

1. **Sending a packet:** For sending data over Ethernet, the user application is the starting point. Irrespective of what application protocol we use (e.g.: MQTT/MQTT-SN), the steps for sending an Ethernet payload are similar as discussed. The first step is to establish a communication path between sender and receiver from the user application. The BSD socket implementation provided by the Zephyr networking stack is used for this purpose. The file descriptor returned after opening a BSD socket (the destination IP address, port number, UDP/TCP as transport type etc are given to the socket) is then used along with the actual payload data to transmit. Note that we can assign several BSD socket options for each opened socket to change the behavior of data transmitted using that socket(e.g. explicitly assign priority to a socket to gain higher priority over other opened sockets). The Zephyr network stack implementation abstracted below the socket layer is responsible to convert the given payload into a TCP/IP or UDP/IP Ethernet packet and hand it over to the device driver for transmission. As shown in Figure 4.1 all the generated Ethernet frames are split into several network packets/buffers (Zephyr maintains a memory pool of fixed-sized network buffers which can be used to store/hold the Ethernet frames until it's transmitted by the driver), and placed into appropriate transmit queues (each transmit queue TX[*], is associated with a transmit thread with a specific priority) according to the socket priority specified. The packets from the high priority queue are taken first and handed it over to the Ethernet driver transmit function (*xmc_eth_tx*) to transmit. The driver will place this data into the XMC Ethernet/MAC specific DMA memory region (TX DMA RAM Buffer) and inform the ETH DMA peripheral to resume the transmit operation. The DMA will transfer this data into the internal 2K MTL TX FIFO of the MAC, and this data is handed over to the PHY via the two transmission lines from the RMII interface of the MAC. A lot of internal processing of the Ethernet data is handled between the MTL FIFO and PHY, like serialising the parallel data, taking transmit timestamp, flow-control, error check (CRC code) and regulating inter-frame gap (IFG), etc. The PHY is responsible for placing the serial bits on to the physical Ethernet cable to transmit.

2. **Receiving a packet:** The receive process starts when the serial data present in the cable is collected by the PHY and filled in the MTL RX FIFO of the XMC Ethernet MAC, through the RMII interface. All the pre-processing of the received data is done before placing the data into MTL FIFO as mentioned

for *Sending a packet*. Also the ETH DMA is triggerrd through the MTL FIFO status fill-level registers. Then its the responsibiliy of the ETH DMA to copy the packet into the XMC Ethenet/MAC specific DMA memory region (RX DMA RAM Buffer) and generate receive event/interrupt flags. The interrupt service routine (ISR) registered for receving events from ETH MAC is *xmc_eth_isr*, and on seeing a receive event from the MAC the *xmc_eth_isr* will signal the driver receive thread (*xmc_eth_rx_thread*) to start processing the Ethernet data available in the RX DMA region. The receive thread will allocate enough network buffer space (from the *fixed size memory pool*) in the Zephyr for the incoming packet and scatter write the complete data across the alloted memory buffers. This buffers are by default placed into the Zephyr RX[0] queue (RX[0] thread). The driver then informs the Zephyr network stack about the successful reception of frames. RX[0] thread pushes this buffers to the upper Zephyr L2 network stack for Ethernet specific packet processing. Later the packets are handed over through IP and TCP/UDP packet processing to be placed into the BSD socket queue for the application program to consume.

4.1.1 Ethernet device driver implementation

The pre-requisites before the actual device driver implementation in Zephyr are describing the hardware and its memory regions to the Zephyr OS eco-system. This is done through writing the devicetree source and bindings for peripherals and memory regions. Defining customised linker scripts to relocate certain memory regions for optimised performance with respect to the XMC memory map etc. The below sections details these implementations.

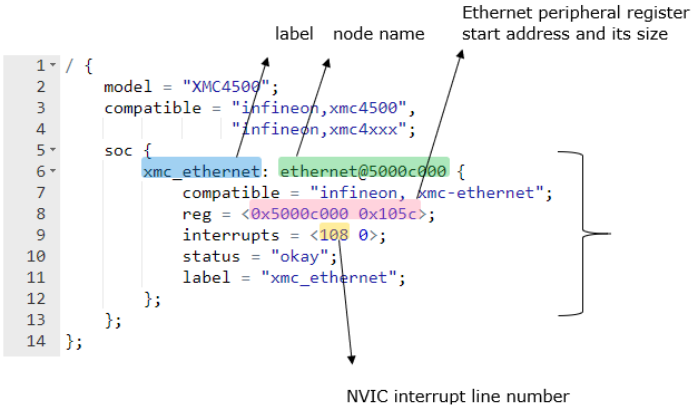


Figure 4.2: Ethernet device tree entry

Zephyr kernel uses "devicetree" implementations to describe hardware in a hirerarcial manner. This described hardware in devicetree format (devicetree source) together with its description as a "binding" is then compiled into a C-header file and passed to the kernel during boot-time. The kernel will load the correspoing driver of the hardware and also initialises related software associated with it.

Figure 4.2 shows the corresponding devicetree source entry done for the XMC 4500 Ethernet driver. The root node is defined as "/" and contains the description of the XMC board. The *model* field describes the SOC model as "XMC4500". The *compatible* field specifies the name of the hardware device the node represents and it can have multiple values. The build system uses the compatible property to find the right bindings for the node. The recommended format is "<vendor,device>" and thus "infineon,xmc4500" is used to specifically identify the node. *Soc* node defined inside the root node contains details of other peripherals present in XMC4500. I've defined a sub-node inside *soc* with 'node name' as *ethernet5000c000*. This node name represents the Ethernet device and also contains the Ethernet peripheral register address space along with its node name (5000c000 is the Ethernet peripheral reg. address). The node label *xmc_ethernet* defined just before the node name can also be used to reference the node later from the driver. Note that the field *reg* inside the Ethernet node has the value pair "<register-start-address size>", which represents the XMC4500 Ethernet MAC peripheral register start address and its size. This information will be later directly queried from the driver to talk with the Ethernet hardware. The field *status* as "okay" tells Zephyr to consider this node as active and load its corresponding driver during bootup. The *interrupt* field with value "108" can be also directly queried from the Ethernet driver during configure phase to connect the NVIC interrupt line with a corresponding driver Interrupt Service Routine (ISR) to receive events/interrupts from the XMC Ethernet hardware.

```
1  description: XMC Ethernet
2
3  compatible: "infineon,xmc-ethernet"
4
5  include: ["ethernet.yaml", "ethernet,fixed-link.yaml"]
6
7  properties:
8      reg:
9        required: true
10     interrupts:
11       required: true
12     label:
13       required: true
```

Figure 4.3: Ethernet device driver yaml file (binding) describing pheripheral properties

Figure 4.3 represents the corresponding devicetree binding of the Ethernet peripheral. The bindings are used to validate the devicetree source file and also to generate the corresponding C-header file together with the source file. The Ethernet binding YAML file describes the contents of the Ethernet node and its correct semantics. The Zephyr build system uses the *compatible* field of both the dts source and binding file ("infineon,xmc-ethernet") to match the correct binding with the source file. The *properties* field describes the mandatory contents to be included in the Ethernet device source node. *include* field will include other generic yaml files with contents/metadata related to Ethernet node from Zephyr.

```
1  config ETHERNET_XMC
2      bool "XMC Ethernet driver"
3      default y
4      help
5        This module implements a kernel device driver for  XMC Ethernet driver.
```

Figure 4.4: Ethernet device driver Kconfig option

Zephyr kernel makes use of the Kconfig file to configure build-time configuration options. I've exposed the option to enable/disable the Ethernet device driver to the user during built-time through the option "ETHERNET_XMC" (as shown in Figure 4.4). By default, its enabled through the boolean option *default* set as "true". Later the CMake function *zephyr_library_sources_ifdef()* exposed by Zephyr (as shown below) is used to include the driver source files (ethernet_xmc.c).

```
zephyr_library_sources_ifdef(CONFIG_ETHERNET_XMC ethernet_xmc.c)
```

```
SECTION_DATA_PROLOGUE(ETH_RAM,(NOLOAD),)  → Name of the output section
{
    . = ALIGN(4);  → 4 byte align
    . = ABSOLUTE(DT_REG_ADDR(DT_NODELABEL(sdram2)));
    KEEP(*(.ETH_RAM));
} GROUP_DATA_LINK_IN(SDRAM2, SDRAM2)
                                  0x30000000
                    output load address
```

Figure 4.5: Linker script to relocate Ethenet DMA memory area to SRAM2

The Infineon XMC4500 provides a high-speed internal Static Random Access Memory (SRAM) section specifically for using with the communication peripherals like Ethernet, USB etc. So I've written a separate linker script file (ETH_RAM.ld) as shown in Figure 4.5, which will place the buffer allocated in the driver for Ethernet DMA operations into the fast SRAM. The linker script gets the start address of this SRAM from the devicetree node "sdram2" defined in the *zephyrproject/zephyr/dts/arm/infineon/xmc4500.dtsi* file. The relevant parts of this file can be seen in Listing 4.1.

The output section name defined in this linker script is *ETH_RAM* as shown in the figure. This 4-byte aligned memory mapping will be linked into the section *SDRAM2*, which is defined in the main Zephyr linker file as shown in Listing 4.2.

```
zephyr_linker_sources(SECTIONS ETH_RAM.ld)
```

The above code defined in the driver CMake file will instruct the build system to include the custom SRAM linker file "ETH_RAM.ld". Later this file is linked with the main Zephyr linker file

```
static XMC_ETH_MAC_DMA_DESC_t ETH_rx_desc[ETH_NUM_RX_BUF]
                    __attribute__((section ("ETH_RAM")));
static uint8_t ETH_rx_buf[ETH_NUM_RX_BUF][XMC_ETH_MAC_BUF_SIZE]
                     __attribute__((section ("ETH_RAM")));
```

The above code snippet shows the allocation of the Ethernet DMA receive descriptors and buffers for the ETH_NUM_RX_BUF number of receive buffers. The *___attribute___((section ("ETH_RAM")))* will force the compiler to allocate the memory region specifically in the ETH_RAM section (i.e. from 0x30000000).

Listing 4.1 gives the minimal nodes defined for the XMC4500 devicetree soc. Line numbers 10-13 declares the fast 32K SRAM region with the label *sdram2*. Node *flash0* is the flash memory to keep data and text regions.*cpus* define the Central Processing Unit (CPU) core type and the *sysclk* give the system clock frequency with which the CPU operates.

Listing 4.2 gives the minimal Zephyr linker script file defined for ARM Cortex M devices (zephyr/include/arch/arm/aarch32/cortex_m/scripts/linker.ld). I've added line number 5 in Listing 4.2, which will reserve the memory region for the SRAM (with section name SDRAM2) in the final Zephyr binary image. Later the ETH_RAM.ld linker file places the code sections with the label "ETH_RAM" into this region.

4.1.1.1 Zephyr Ethernet driver instantiation flow

Zephyr device driver can create the device from the devicetree in two ways: (i) Using instance numbers and (ii) Using devicetree node labels. Instance based APIs are used by (i) which can automatically instantiate devices with instance numbers. This method only requires that the *compatible* property of the devicetree node be equivalent to the one defined in the source. I've chosen this method over (ii) because it doesn't need manual instantiation by referring to the node label.

Listing 4.1: The generic devicetree include file defining the XMC4500

```
/ {
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu0: cpu@0 {
            compatible = "arm,cortex-m4f";
            reg = <0>;
        };
    };
    sdram2: memory@30000000 {
        compatible = "mmio-sram";
        reg = <0x30000000 DT_SIZE_K(32)>;
    };
    flash0: serial-flash@c000000 {
        compatible = "serial-flash";
    };
    sysclk: system-clock {
        compatible = "fixed-clock";
        clock-frequency = <120000000>;
        #clock-cells = <0>;
    };
};
```

Listing 4.2: Linker file definition for ARM Cortex M cores

```
MEMORY
{
    FLASH (rx) : ORIGIN = ROM_ADDR, LENGTH = ROM_SIZE
    SRAM  (wx) : ORIGIN = RAM_ADDR, LENGTH = RAM_SIZE
    LINKER_DT_REGION_FROM_NODE(DT_NODELABEL(sdram2), rw)
    IDT_LIST (wx) : ORIGIN = 0xFFFFF7FF, LENGTH = 2K
}
```

```
#define DT_DRV_COMPAT infineon_xmc_ethernet
```

The above define is added at the top of the Ethernet driver source file, to make it equivalent to the *compatible* of the devicetree node (which was "infineon,xmc-ethernet"). Note that all the special characters (here ',' and '-') are replaced by '_' for the define.

```
DT_INST_FOREACH_STATUS_OKAY(XMC_ETH_INIT)
```

The above instance based API added to the driver source is used to access the devicetree node data. Here we check if the Ethernet node in the devicetree has "okay" for the *status*. If it's true then the instantiation macro "XMC_ETH_INIT" defined at the end of the source code (Listing 4.3) is called for the actual device instantiation.

Listing 4.3: Device driver instantiation macro

```
#define XMC_ETH_INIT(n)                                                    \
    static void eth##n##_config_func(void) {                               \
            IRQ_CONNECT(DT_INST_IRQN(n),                                   \
                DT_INST_IRQ(n, priority), xmc_eth_isr,                     \
                DEVICE_DT_INST_GET(n), 0);                                 \
            irq_enable(DT_INST_IRQN(n)); }                                 \
                                                                           \
    static struct eth_context eth##n##_context = {                         \
        .eth_mac = {.regs = (ETH_GLOBAL_TypeDef *)                         \
                        DT_INST_REG_ADDR(n),                               \
                    .address = MAC_ADDR,                                   \
                    .rx_desc = ETH_LWIP_0_rx_desc,                         \
                    .tx_desc = ETH_LWIP_0_tx_desc,                         \
                    .rx_buf = &ETH_LWIP_0_rx_buf[0][0],                    \
                    .tx_buf = &ETH_LWIP_0_tx_buf[0][0],                    \
                    .num_rx_buf = ETH_LWIP_0_NUM_RX_BUF,                   \
                    .num_tx_buf = ETH_LWIP_0_NUM_TX_BUF},                  \
        .eth_phy_config = { .interface =                                   \
                        XMC_ETH_LINK_INTERFACE_RMII,                       \
                            .enable_auto_negotiate = true},                \
        .config_func = eth##n##_config_func,                               \
        .phy_addr = 0U,                                                    \
        .mac_addr = {MAC_ADDR0, MAC_ADDR1, MAC_ADDR2,                      \
                     MAC_ADDR3, MAC_ADDR4, MAC_ADDR5}                      \
        };                                                                 \
                                                                           \
    ETH_NET_DEVICE_DT_INST_DEFINE(n, xmc_eth_init, NULL,                   \
        &eth##n##_context, NULL, CONFIG_ETH_INIT_PRIORITY,                 \
        &api_funcs, NET_ETH_MTU);                                          \
```

Listing 4.3 shows a glimpse of how the XMC Ethernet driver instantiation macro is defined. *XMC_ETH_INIT* receives each instance number of the device (devicetree node with *status="okay"*) as its argument. This instance number is passed along

to the *ETH_NET_DEVICE_DT_INST_DEFINE* API (line 27-29) to create an Ethernet network interface and bind it to network device (by taking metadata from the devicetree).

The function which will initialise the driver (*xmc_eth_init*), pointer to devices private data (*&eth##n##_context*), the boot-time init priority of the driver (*CONFIG_ETH_INIT_PRIORITY*), pointer to the API function struct used by the driver (*&api_funcs*) and the maximum transfer units (bytes) for this network interface etc are given to *ETH_NET_DEVICE_DT_INST_DEFINE*. *&eth##n##_context* is the instance of *eth_context* structure which holds all the private data of the driver like the MAC address, DMA descriptors and buffers, the configuration function (*eth##n##_config_func*) etc. *eth##n##_config_func* is responsible for collecting the Ethernet interrupt line number from the devicetree and connecting the ISR routine (*xmc_eth_isr*) with it.

The Zephyr device model provides a consistent method for configuring all the drivers that are part of the system. This model is also responsible for initialising all the configured drivers. Zephyr exposes generic APIs (function pointers) for each subsystem type (Ethernet, UART etc). It's up to the board vendors to implement their board-specific drivers and fill the generic function pointers with the implementations. Listing 4.4 shows all the XMC4500 specific Ethernet driver implementations (*xmc_eth_**) that are being assigned to the Zephyr generic APIs (*struct ethernet_api* representing Ethernet subsystem). The implementations for the main APIs like ethernet_api.iface_api.init and ethernet_api.send are explained in the following sections.

All the important Ethernet driver functionalities are explained in the following section with its flow diagrams.

4.1.1.2 Ethernet init function

Figure 4.6 represents the Ethernet device driver init function (*xmc_eth_init ()*) which will be called first when the driver loads. Step 1, initialises the RMII interface of the MAC through which the MAC exchanges Ethernet data between PHY, once it becomes operational. The Station Management Agent (SMA) present in MAC defines a 2-wire protocol interface called Station Management Interface (SMI), which is used initially to set up the PHY with configuration by accessing its registers. SMI defines two lines named: Management Data Input/Output (MDIO) line for sending/receiving data and Management Data Clock (MDC) for a clock. These modules are also initialised

Listing 4.4: Zephyr Ethernet subsystem APIs assigned with xmc-specific driver implementations

```
static const struct ethernet_api api_funcs = {
    .iface_api.init        = xmc_eth_iface_init,
    .get_capabilities    = xmc_eth_get_capabilities,
    .set_config        = xmc_eth_set_config,
    .get_config        = xmc_eth_get_config,
    .send            = xmc_eth_tx,
    .start             = xmc_eth_start,
    .stop            = xmc_eth_stop,
#if defined(CONFIG_PTP_CLOCK_XMC)
    .get_ptp_clock = xmc_eth_get_ptp_clock,
#endif
};
```

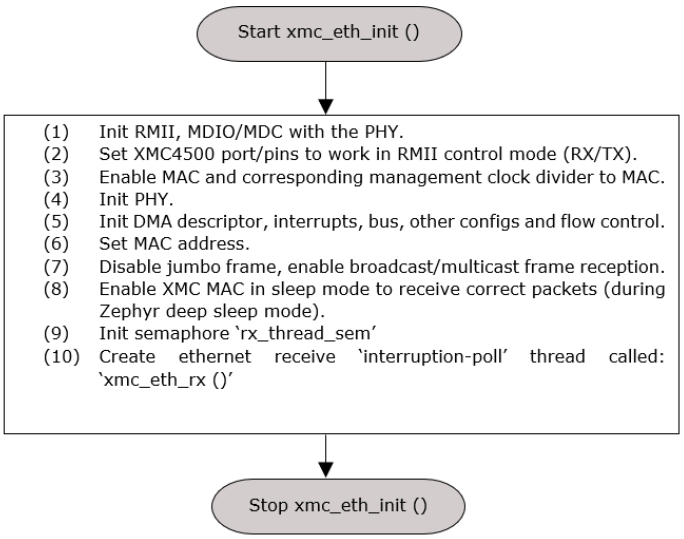


Figure 4.6: xmc_eth_init() function

during this step. Step 2 changes the port function of the XMC4500 to work with the RMII interface pins. The XMC MAC IP core and its management clock dividers are initialised during step 3, followed by initialising PHY in step 4.

The Ethernet MAC DMA descriptors, interrupts, flow control configuration, the AHB bus burst mode etc are configured during step 5. After setting the MAC address given by the driver in step 6, the function proceeds to set multicast/broadcast filters in the MAC, along with disabling jumbo frame receptions. Since Zephyr support

deep-sleep mode during Idle task scheduling to save power consumption, we need to enable the MAC peripheral to stay awake even when the CPU enters a sleep mode (triggered by the kernel) in step 8. Step 9 inits the *rx_thread_sem* semaphore which is used by the ISR (xmc_eth_isr ()) to signal the driver receive thread to read packets. Step 10 does the actual creation of the interrupt-poll receive thread called *xmc_eth_rx ()*, which will be waiting on *rx_thread_sem* to read/copy incoming packets from the RX DMA Random-Access Memory (RAM) buffer, when a receive event is triggered.

4.1.1.3 Ethernet receive thread

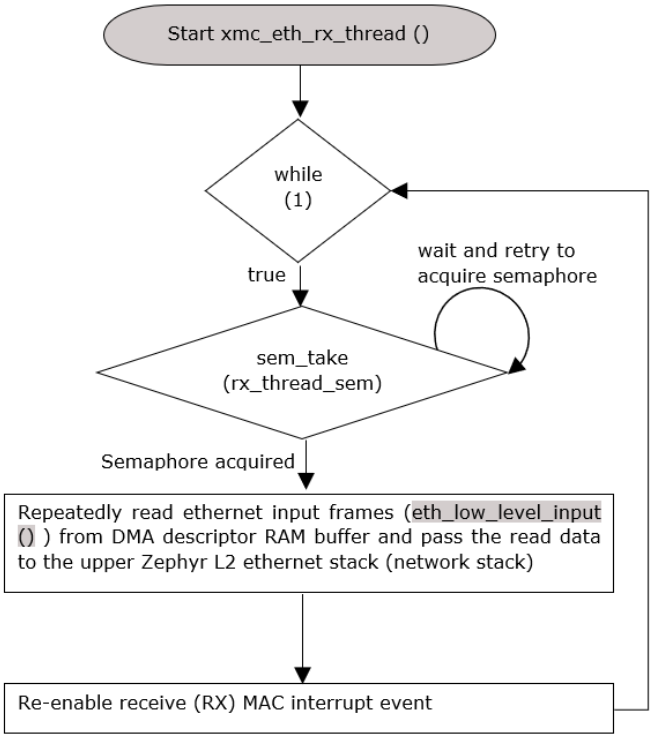


Figure 4.7: xmc_eth_rx_thread

The driver receive thread which behaves as an interrupt-polling thread is depicted in Figure 4.7. As shown in the flow diagram, the outer while(1) loop makes it always polling to read the data from the DMA RX RAM buffer region. The only thing blocking it from actually reading the data using the *eth_low_level_input ()* is the *rx_thread_sem* semaphore. The ISR registered with the MAC (*xmc_eth_isr*) needs to be triggered first for the *xmc_eth_rx_thread ()* to acquire the semaphore and consequently read the incoming packets residing in the DMA memory. Note that

during the repeated reading of packets from DMA, the receive of MAC interrupt events are temporarily disabled by the *xmc_eth_isr*. This needs to be re-enabled after the read operation.

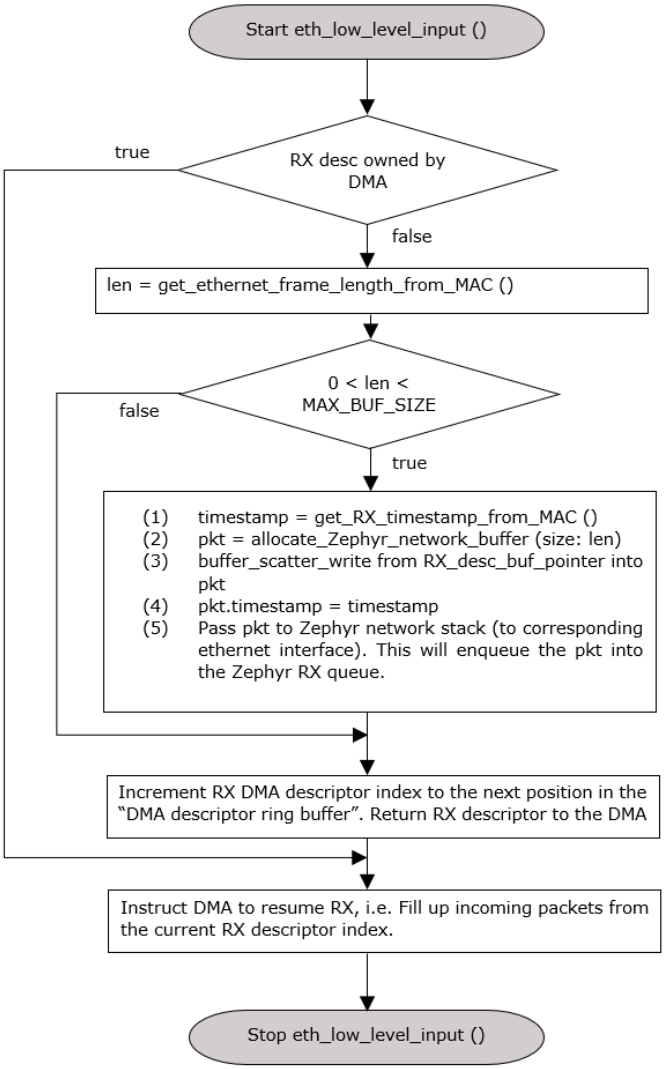


Figure 4.8: xmc_eth_low_level_input() function

The actual control flow of reading incoming packets once a receive interrupt is triggered is depicted in Figure 4.8. We can only access the RX DMA memory buffer if the corresponding status word of the receive descriptor says so. That is if the descriptor is not owned by the DMA, we can go ahead and read. Once this condition is satisfied we make sure the length of the received Ethernet frame residing in the DMA is within maximum limits. Later enough Zephyr network buffers are allocated

and the read data is scatter written into them, along with the receive timestamp queried from the MAC. Then we inform the Zephyr network stack about the successful receive event for L2 processing, so that the OS enqueues the received packets into the Zephyr RX queue. XMC DMA descriptors are structured in a ring-like fashion, similar to single-linked list. The driver ha s to increment its index and inform the MAC to resume RX operation, so that new data is filled in the next available DMA buffer location.

4.1.1.4 Ethernet interrupt service routine

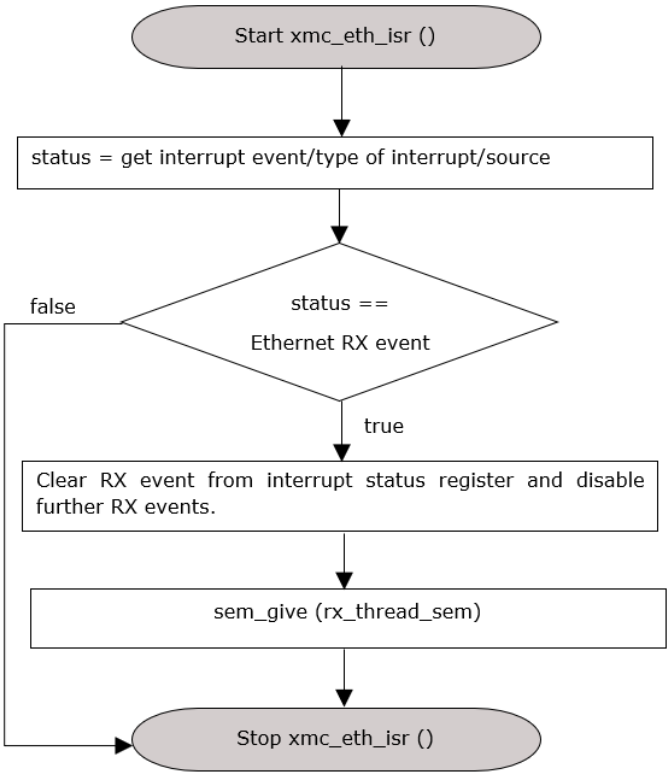


Figure 4.9: xmc_eth_isr() function

Figure 4.9 is the flow of the ISR registered with the MAC to receive interrupts. On receiving an RX event, the routine will clear the interrupt status word and disables further receive events temporarily till the current incoming packets are serviced. Later releasing/giving the *rx_thread_sem* semaphore will signal the *xmc_eth_rx_thread* which has been waiting on the same semaphore to repeatedly read the packets out of DMA buffers.

4.1.1.5 Ethernet send function

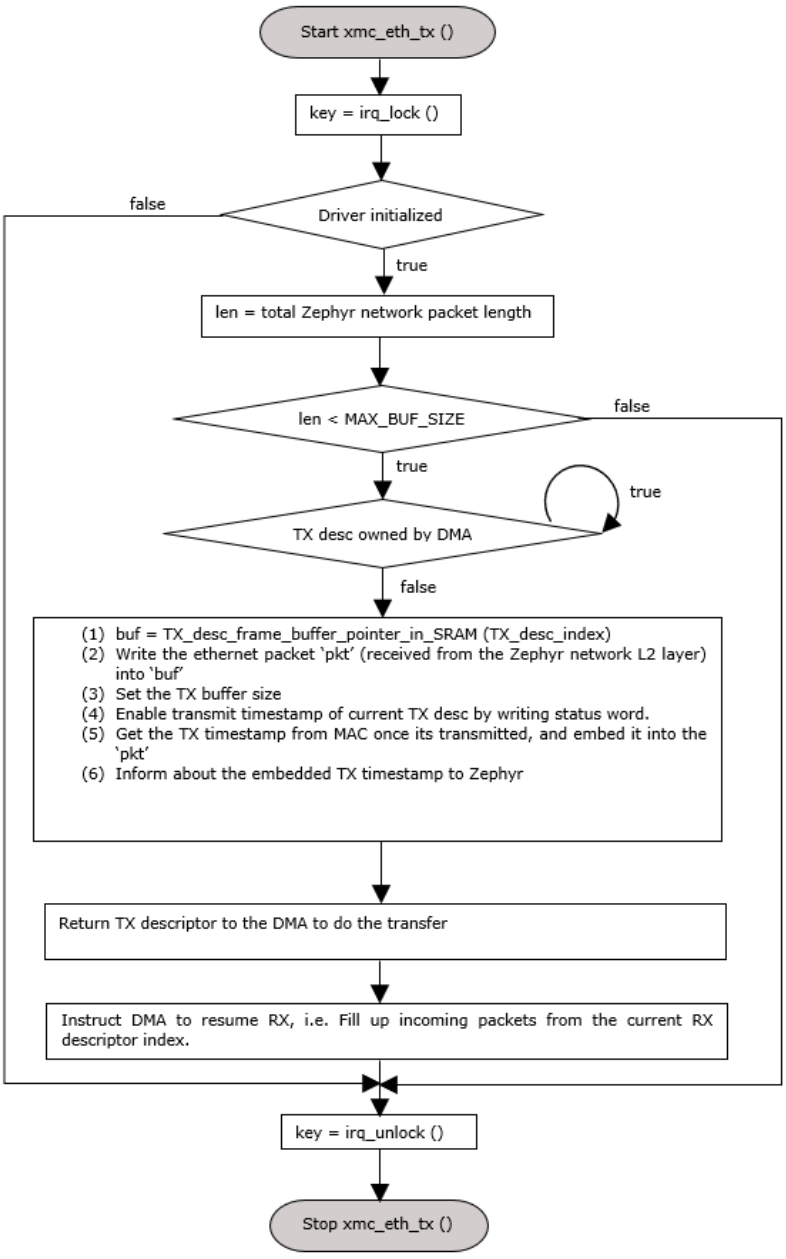


Figure 4.10: xmc_eth_tx() function

The *xmc_eth_tx* function shown in Figure 4.10 is used to assign the *ethernet_api.send* function pointer of the Ethernet driver. The Zephyr L2 network stack will call this assigned function whenever a packet needs to be send from the internal

TX queues. Basic sanity check such as if the driver is already initialised and if the passed network packets from Zephyr is small enough to fit into the DMA buffers etc. are done before proceeding. As mentioned in the *eth_low_level_input*, we should make sure that the TX DMA descriptors are not owned by DMA before actually writing the Zephyr network packets into the DMA TX buffer region. After the packet is written, its length should be set for the transfer. Enabling the MAC Intellectual Property (IP) timestamp module and collecting the generated timestamp to embed into the Zephyr network packet is done in next stage. We need to inform the Zephyr stack about successful timestamp collection after embedding the information into the packet, so that the stack could perform PTP related operations with it. As mentioned in the *eth_low_level_input* the TX DMA descriptors are released and we instruct the DMA to resume its operations after all the steps.

4.1.1.6 Ethernet interface init funcion

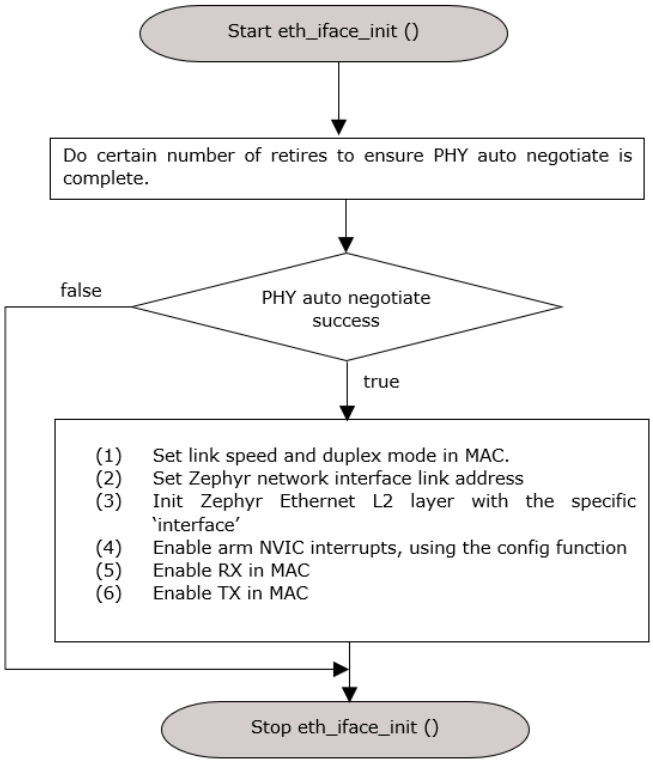


Figure 4.11: xmc_eth_iface_init() function

The *eth_iface_init ()* function depicted in Figure 4.11 is used to assign the *ethernet_api.iface_api.init* function pointer of the Ethernet driver. This will be called

after the Ethernet driver is initialised with *xmc_eth_init ()* to initialise the Ethernet network interface. Before proceeding further into the function we must ensure that the PHY is initialised and ready to be used. Then we set the duplex mode as full duplex and the link speed as 100 Mbit/s. We'll set the link address with the MAC address and initialises the Zephyr network L2 interface with this specific interface. Later the Ethernet driver config function (*eth_config_func ()*) is called to enable the NVIC Ethernet MAC interrupts and assign the ISR routine (*xmc_eth_isr ()*) to that interrupt line. Finally the receive and transmit functionalities are enabled to set the Ethernet driver fully functional with the hardware.

4.2 Network time synchronisation

The PTP driver implemented in Zephyr RTOS for Infineon XMC4500 microcontroller provides support for network time synchronisation. Implementation details of the driver are given in the following sections.

4.2.1 PTP driver implementation

Network time synchronisation needed for implementing advanced traffic shapers to enable deterministic communication is provided by the gPTP (IEEE 802.1AS) stack implemented in Zephyr RTOS. The Zephyr config option *CONFIG_NET_GPTP* will enable this subsystem and initialises all the PTP related state machines as defined in the IEEE 802.1AS-2011 standard and its communication.

```
6  config PTP_CLOCK_XMC
7      bool "XMC Ethernet ptp driver"
8      default y
9      help
10     This module implements a kernel ptp device driver for XMC Ethernet driver.
11
```

Figure 4.12: PTP device driver Kconfig option

The configuration option *PTP_CLOCK_XMC* is exposed from the build system to the user for enabling or disabling the PTP device driver as shown in Figure 4.12. By default the PTP driver is enabled and the kernel will load and initialises the driver during boot-up.

Listing 4.5 shows the macro which is used to create the PTP device object and set it up for boot time initialisation. Since the PTP device is a part of the Ethernet

Listing 4.5: PTP device driver instantiation macro

```
1 DEVICE_DEFINE (xmc_ptp_clock, PTP_CLOCK_NAME, ptp_xmc_init,
2           NULL, &ptp_xmc_context, NULL, POST_KERNEL,
3           85, &ptp_api_implementation);
```

device, we are not allocating the PTP device from a devicetree node. Hence I've used *DEVICE_DEFINE()* macro to create the device. The kernel will configure the *struct device* object defined by *DEVICE_DEFINE()*, specifically for the PTP during system initialisation.

The function which will initialise the driver (*ptp_xmc_init*), pointer to devices private mutable data (*&ptp_xmc_context*), the boot-time init priority of the driver (85, which tells the kernel to initialise PTP before the network stack with priority of 90), pointer to the API function struct used by the driver (*&ptp_api_implementation*) and the device initialisation level (*POST_KERNEL*) etc are given to *DEVICE_DEFINE()*. *&ptp_xmc_context* is the instance of *ptp_context* structure which holds all the private mutable data of the PTP driver like the reference to the instance of Ethernet driver (*struct eth_context*).

4.2.1.1 PTP driver init function

The function pointer *ptp_xmc_init* passed to the *DEVICE_DEFINE()* macro is used by the kernel to initialise the PTP device during system initialisation. The main three functionalities of *ptp_xmc_init* are: (i) Zephyr device context init, (ii) XMC PTP hardware init and (iii) initialisation of PTP accuracy measurement logic.

The Ethernet driver instance structure (*struct eth_context*) has a field dedicated to be pointed to the PTP driver instance. Similarly, the PTP driver instance structure (*struct ptp_context*) has a field dedicated to be pointed to the Ethernet driver instance. This cyclic reference to-and-from PTP and Ethernet drivers are done in (i).

During (ii) XMC PTP hardware init, the initial values for the PTP system clock are assigned. As described in Section 2.6.1, the *sub-second increment* register (*SUB_SECOND_INCREMENT*) and *timestamp addend* register (*TIMESTAMP_ADDEND*) are also assigned here, for the proper working of the PTP clock.

The maximum precision with which the XMC PTP hardware can count a second is with 20 ns. Hence the *SUB_SECOND_INCREMENT* is assigned with an initial

value of *20*. Thus a PTP clock frequency of 50 MHz is needed to count a complete second (50 MHz x 20 ns = 1 s). The Ethernet PTP peripheral clock is derived from the XMC4500 processor system clock having a frequency of 120 MHz. The relation between the PTP clock and the main system clock can be stated as: 50 MHz ticks of PTP is equivalent to 120 MHz ticks of the system clock to achieve a second. Hence a *clock-ratio* of (120 MHz/50 MHz) = *2.4* ticks of the system clock is needed to complete a single PTP tick (equivalent to one 20 ns step).

The PTP frequency of 50 MHz is achieved by setting the *TIMESTAMP_ADDEND* register with a specific value such that, it'll be repeatedly added into an *accumulator* register (32-bit). The *Sub-second register* of PTP is incremented when this 32-bit *accumulator* register overflows. Hence a value of (2^{32}/2.4) should be written to the *TIMESTAMP_ADDEND* register, such that the *accumulator* register overflows in *2.4* ticks of the system clock and hence the *Sub-second register* increments by one (meaning 20 ns have passed).

The final part of the *ptp_xmc_init* initialises General Purpose Input Output (GPIO) pins to be toggled every PTP second. The XMC PTP hardware provides an *alarm* feature by which an interrupt is generated when the specified time in *second:nanoseconds* have expired. This *alarm* feature is used to trigger an interrupt every other second, so that the GPIO pin can be toggled, to measure PTP accuracy.

Listing 4.6: Zephyr PTP subsystem APIs assigned with xmc-specific driver implementations

```
1
2 static const struct ptp_clock_driver_api ptp_api_implementation = {
3     .set = ptp_clock_xmc_set,
4     .get = ptp_clock_xmc_get,
5     .adjust = ptp_clock_xmc_adjust,
6     .rate_adjust = ptp_clock_xmc_rate_adjust,
7 };
```

The PTP subsystem driver API structure *ptp_clock_driver_api* needs to be implemented by the board vendor (e.g. Infineon XMC4500) according to their PTP hardware to utilise the Zephyr gPTP stack. The XMC4500 implementation done for the generic Zephyr PTP driver subsystem APIs is shown in Listing 4.6. A brief overview of individual implementations are given below.

4.2.1.2 PTP clock set function

The time of PTP clock is set inside this function (*ptp_clock_xmc_set ()*). Zephyr gPTP stack calculates the time to set in *seconds:nanoseconds* format and calls this function to set the same in PTP XMC hardware.

4.2.1.3 PTP clock get function

The current time of the XMC PTP clock is retrieved and returned from this function (*ptp_clock_xmc_get ()*). Zephyr gPTP stack calls this driver function to calculate the time delta that is needed to be adjusted for time synchronisation.

4.2.1.4 PTP clock adjust function

The time in *nanoseconds* is received from the Zephyr gPTP stack as an argument to *ptp_clock_xmc_adjust ()*. *ptp_clock_xmc_adjust ()* is responsible for resetting the nanoseconds value of the XMC PTP clock with the received value. This driver function will be generally called when the Zephyr gPTP stack needs to adjust the hardware PTP clock with a large value immediately.

4.2.1.5 PTP rate adjust function

Unlike the *ptp_clock_xmc_adjust ()*, the PTP rate adjust function *ptp_clock_xmc_rate_adjust ()* will adjust the XMC PTP clock time change rate. Zephyr gPTP stack calls *ptp_clock_xmc_rate_adjust ()* with a *ratio* argument. This *ratio* represents the percentage by which the rate of PTP clock time change needed to be increased or decreased. This is achieved by updating the *TIMESTAMP_ADDEND* register with the recalculated addend value based on the received *ratio*.

4.3 Paho MQTT-SN Client library integration into Zephyr RTOS

Eclipse Paho [6] is the open-source implementation of the MQTT-SN protocol by Eclipse specifically targetted at embedded platforms. The Paho client library is lightweight and uses very limited resources. The library is built in such a way that the network transport layer, memory management layer, etc. are separated from the core MQTT-SN serialisation/deserialisation code.

To support MQTT-SN transmit and receive from the Zephyr RTOS, the Paho *MQTTSNPacket/* source is ported to Zephyr RTOS by using the *CMake* build system. The control applications running on the *control nodes* (Zephyr RTOS on XMC4500) can use the APIs provided by the *MQTTSNPacket* for sending payloads in MQTT-SN format. Sample APIs of this source is given in Listing 4.7.

Listing 4.7: Sample APIs of core *MQTTSNPacket* source

```
1 //Serialisation APIs needed to convert into MQTT-SN packet structure
2 MQTTSNSerialize_connect ()
3 MQTTSNSerialize_register ()
4 MQTTSNSerialize_publish ()
5
6 //Deserialisation APIs needed after read a received MQTT-SN packet
7 MQTTSNDeserialize_puback ()
8 MQTTSNDeserialize_regack ()
9 MQTTSNDeserialize_connack ()
```

A separate *MQTTSNTransport* code structure is implemented where the BSD socket network transport layer is ported to support Zephyr RTOS implementation. The main functionalities of this layer include socket *open*, *close*, *sending packet buffer*, *bind to UDP port*, *receiving packet buffer*, etc. *MQTTSNTransport* is used by the control applications to transmit *MQTTSNPacket*-serialised data. Similarly *MQTTSNTransport* is used for receiving MQTT-SN packets.

5

Experimental study

All the software based implementations and tools used along with the hardware setup and measurement logic adopted for conducting the experimental study for the thesis is explained in this chapter.

5.1 Software setup for measurements

5.1.1 Zephyr application for measurements

Section 5.1.1.1 and Section 5.1.1.2 illustrates the overview of publisher/subscriber applications implemented in Zephyr RTOS for publishing and receiving M2M payloads on a specific topic (for MQTT and MQTT-SN). The measurement logic explained in Section 5.1.2.1 is then used to analyse the GPIO toggled waveforms collected from both publisher and subscriber. The analysed results from the resulting pulses give us the latency and jitter distribution for comparing the performance of MQTT and MQTT-SN (summarised in Section 6.1).

5.1.1.1 MQTT Publisher/Subscriber implementations

Figure 5.1 shows the M2M communication flow from Node A (MQTT publisher on Zephyr RTOS/XMC4500) to Node B (MQTT subscriber on Zephyr RTOS/XMC4500) connected through a switch. A host system (Node C) running an MQTT broker (eg: mosquitto on Ubuntu/ThinkPad E14) is present on the network to manage and route payloads (from publisher to subscriber) based on topic subscriptions. All the nodes are in the same network (same network address for IP) and the publish event is represented by (1), followed by (2) to represent the receive event (of the corresponding published packet).

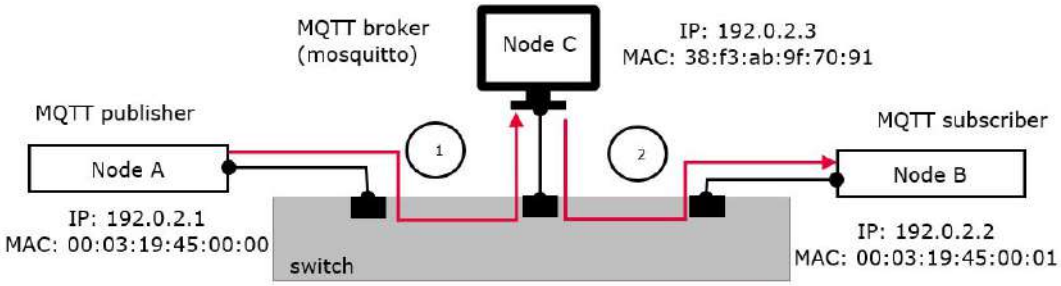


Figure 5.1: MQTT communication flow where 'Node A'/publisher publishing topics to 'Node C'/'MQTT broker on host' (1), and Node C republishing that to 'Node B'/subscriber (2).

The MQTT publisher application implementation on Node A is represented in Figure 5.2. The MQTT network APIs exposed by Zephyr RTOS is used to write the publisher. After initialising the GPIOs, I'll try to connect (MP.2) to the MQTT broker (Node C in Figure 5.1), and its steps are given below:
try_to_connect() : Here I'll initialise the 'mqtt_client' structure exposed by Zephyr with the broker IP/port address, 'mqtt_event_handler ()' as the MQTT event handler callback function (to service different MQTT events received from the broker), client ID string, MQTT protocol version and TX/RX buffer memory in application space. The initialised 'mqtt_client' structure variable is used along with the 'mqtt_connect()' API to establish a connection to the broker. When the connection is successful I'll wait/poll on the opened TCP socket file-descriptor in 'mqtt_client' to receive a read event to trigger a 'mqtt_input()'. 'mqtt_input()' manages and processess incoming MQTT traffic and triggers the assigned event callback 'mqtt_event_handler ()' with 'MQTT_EVT_CONNACK', so that we can set our application state variable as "connected".
'process_mqtt_and_sleep ()' shown in MP.4 has the sole function of keeping the established MQTT connection alive by sending MQTT live packets and processing incoming MQTT packets/replies. Next step (MP.5) the MQTT payload structure is initialised with topic string (eg: "sensors"), payload data (i've used 16 Byte string data), its corresponding lengths, the QoS (0 - At most once, 1 - At least once or 2 - Exactly once) level needed for the publish along with other flags. To publish 500 (shown as MAX_PUBLISH in MP.6) MQTT packets, we repeatedly transmit the assigned payload structure(from MP.5) using 'mqtt_publish()' as shown in MP.6 and MP.7. GPIO pins are toggled immediately after successful publish along with the call to 'process_mqtt_and_sleep ()' to receive and service the incoming MQTT publish acknowledgment events from the broker (MQTT_EVT_PUBACK, MQTT_EVT_PUBREC and MQTT_EVT_PUBCOMP will be received based on

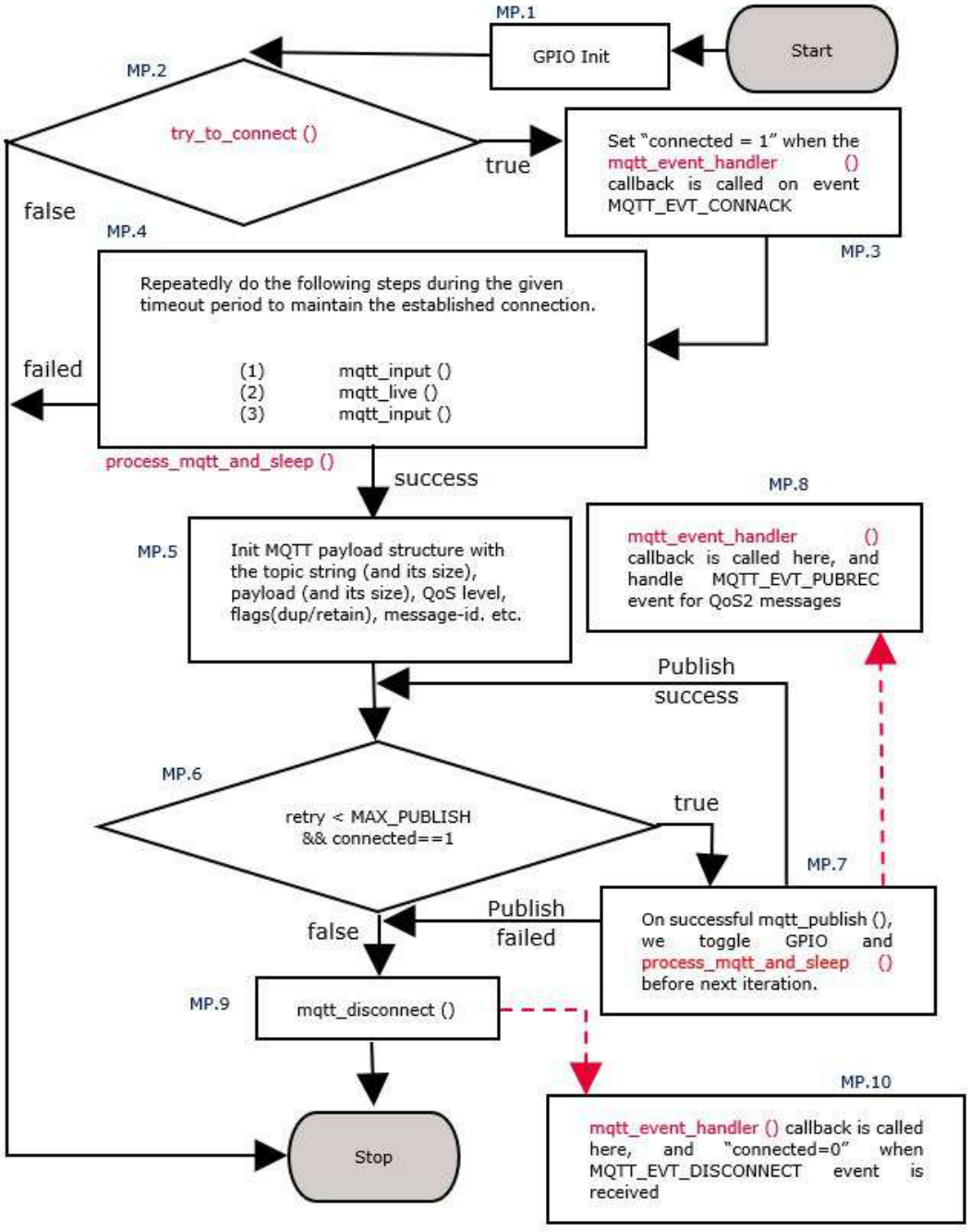


Figure 5.2: MQTT publisher application flowchart on ZephyrRTOS

the published QoS level). mqtt_disconnect() is called to end the MQTT connection after all the publishes as shown in MP.9. Note that after each mqtt_publish() and

mqtt_disconnect() api calls, the registered callback 'mqtt_event_handler ()' is triggered from the broker with the corresponding event (MQTT_EVT_*) so as to inform the publisher about acknowledgements/state changes.

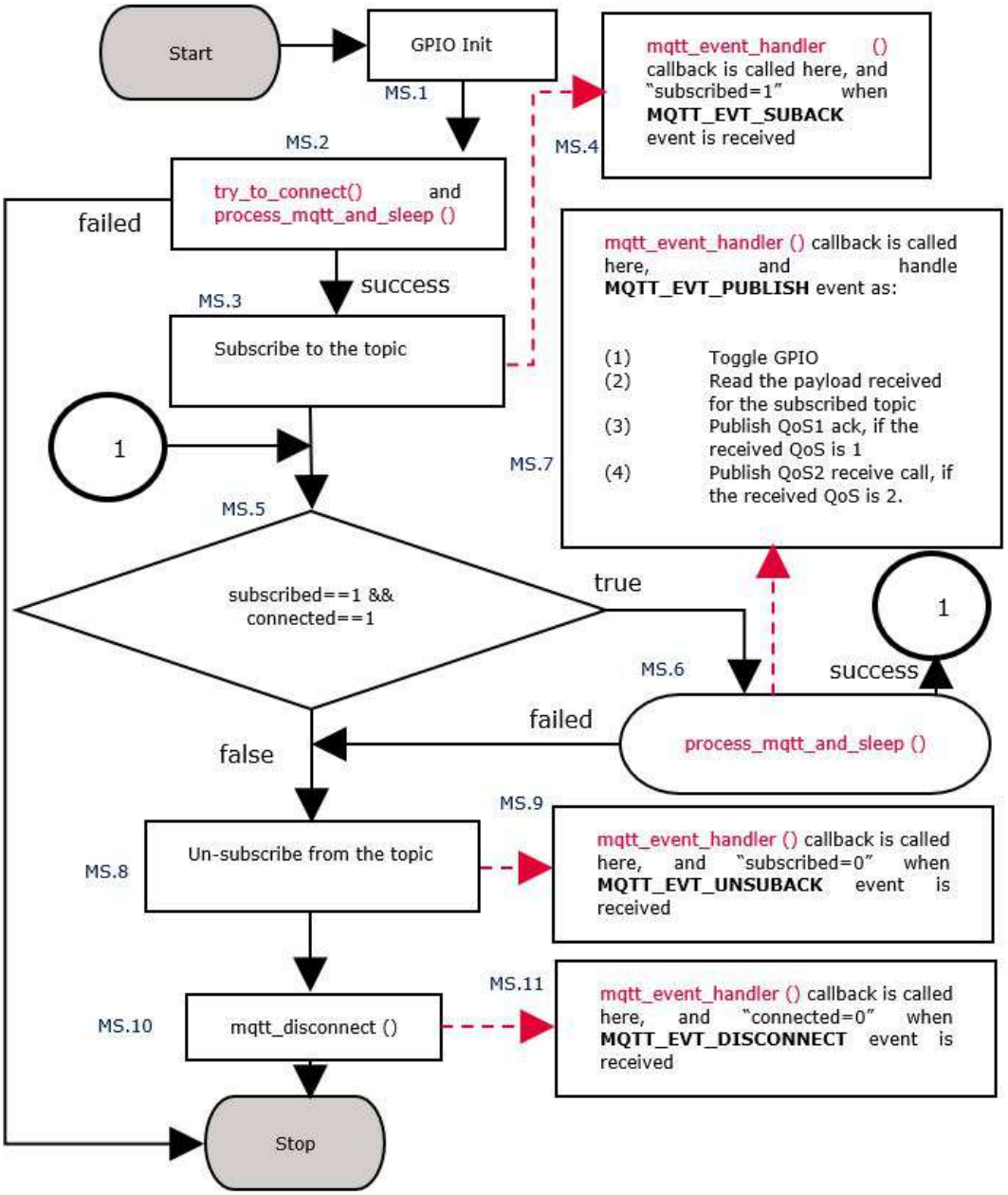


Figure 5.3: MQTT subscriber application flowchart on ZephyrRTOS

The MQTT subscriber application implementation on Node B is represented in Figure 5.3. The MQTT network APIs exposed by Zephyr RTOS is used to write the

subscriber. After initialising the GPIOs, i'll try to connect (MS.2) to the MQTT broker (Node C in Figure 5.1) as mentioned before in publisher, and the corresponding event handler ('mqtt_event_handler ()') with the connection acknowledgment('MQTT_EVT_CONNACK') is called on success.

Next step is to subscribe (MS.3) to a topic (eg: "sensors") with the broker, to receive its payloads. As mentioned in the publisher section, the 'mqtt_client' structure used to establish connection is used with 'mqtt_subscribe ()' api, along with the subscription topics and its QoS level to subscribe. On successful subscription, the broker triggers 'mqtt_event_handler ()' with the 'MQTT_EVT_SUBACK' event as shown in MS.4. We assign our internal subscription state as valid in this received callback.

The busy loop where we are polling for MQTT publish events and its servicing are represented in MS.5, MS.6 and MS.7. I'll toggle the GPIO from the callback 'mqtt_event_handler ()' on seeing a publish event (MQTT_EVT_PUBLISH) after processing the received payload. The corresponding handshake MQTT replies needed to be published for different QoS levels are also taken care in this section. After successful reception of 500 MQTT publish packets the cleanup procedure like unsubscribing (MS.8) and disconnecting (MS.10) are done before stopping the application.

5.1.1.2 MQTT-SN Publisher/Subscriber implementations

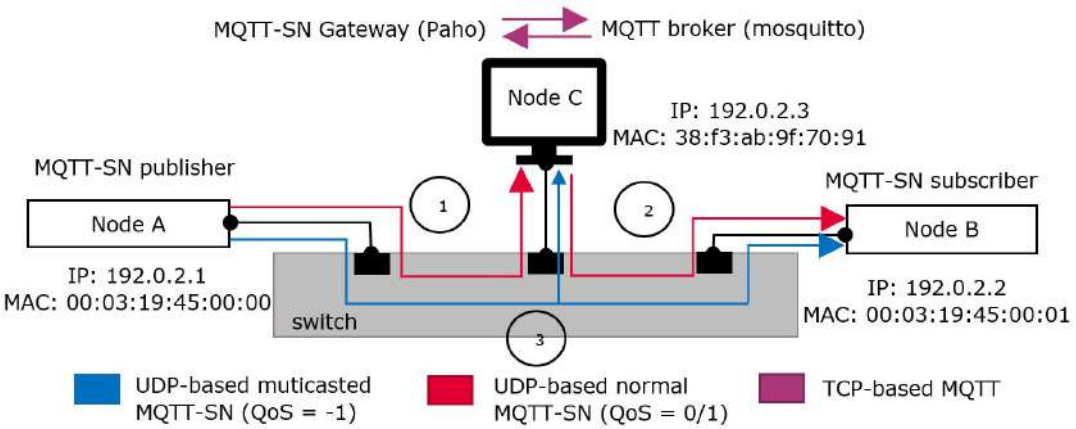


Figure 5.4: MQTT-SN communication flow for publishing a message with different QoS levels.

Figure 5.4 shows the M2M communication between Node A (MQTT-SN publisher) and Node B (MQTT-SN subscriber) for different MQTT-SN QoS levels. Node C

connected to the same IP network is listening on its UDP port (eg: 192.0.2.3:10000) for incoming MQTT-SN packets. These UDP-based MQTT-SN publish packets (blue and red arrows from A to C) will be first received by the MQTT-SN Gateway (i've used the opensource Eclipse 'Paho'), which will process and translate the "UDP-based MQTT-SN" traffic into "TCP-based MQTT", and routed towards the MQTT broker (Eclipse mosquitto) running on the same host (the flow shown as purple arrow). The broker will reply with a "TCP-based MQTT " publish packet back to the gateway (intended for the corresponding subscribed node), which will be again converted back into "UDP-based MQTT-SN" before forwarding it to the subscriber, node B (as shown with red arrow from C to B).

It's important to note that, not all the MQTT-SN publish packets (from Node A) need to be first routed towards the MQTT-SN Gateway (Node C) before being received by Node B. If publishing with QoS -1 (fire-and-forget) packets, it can be directly sent from source to destination without any translation/ processing/ topic-management in-between. This flexibility/less-overhead in publishing can be further adapted to serve multiple receive nodes by sending to a multicast address. So that only the receive nodes joined in this multicast group will receive the corresponding publish packet. Here subscribing to a topic is avoided by just joining to the multicast address and the topic management done by the broker is delegated/simplified for the switch to forward the multicasted-QoS-1-publish packets into the multicast group. This is represented by the blue arrow in the Figure 5.4. In this way i've mimicked the complete M2M topic management done together by Gateway/broker with just a switch multicast concept and avoided the extra latency/delay associated with it. The Gateway residing in the Node C (host system) can be also part of the same multicast group to be a passive listener to forward the publish packets to the outside world (eg: for logging) to scale-up.

The MQTT-SN publisher application residing in Node A is represented in Figure 5.5. The open-sourced "Eclipse Paho MQTT-SN C/C++ client for Embedded platforms" code is integrated into the Zephyr RTOS, and the BSD sockets (UDP) opened to the MQTT-SN Gateway node (Node C) is used with the library to implement the publisher. After we initialise GPIOs (MNP.1), the UDP socket opened in the non-blocking mode (MNP.2) is used as the transport layer for MQTT-SN publisher client. The publisher is connected to the Gateway/broker only if its QoS is greater than -1 (using connect_to_mqtt_sn_gateway () as shown in MNP.4). ***connect_to_mqtt_sn_gateway ()*** : First a MQTT-SN specific connection structure needs to be initialised with "client-id" string and other options, before converting it into a serial byte buffer (by making use of the MQTT-SN specific connection serialisation api exposed by Paho). This serialised data buffer is then directly transmitted to

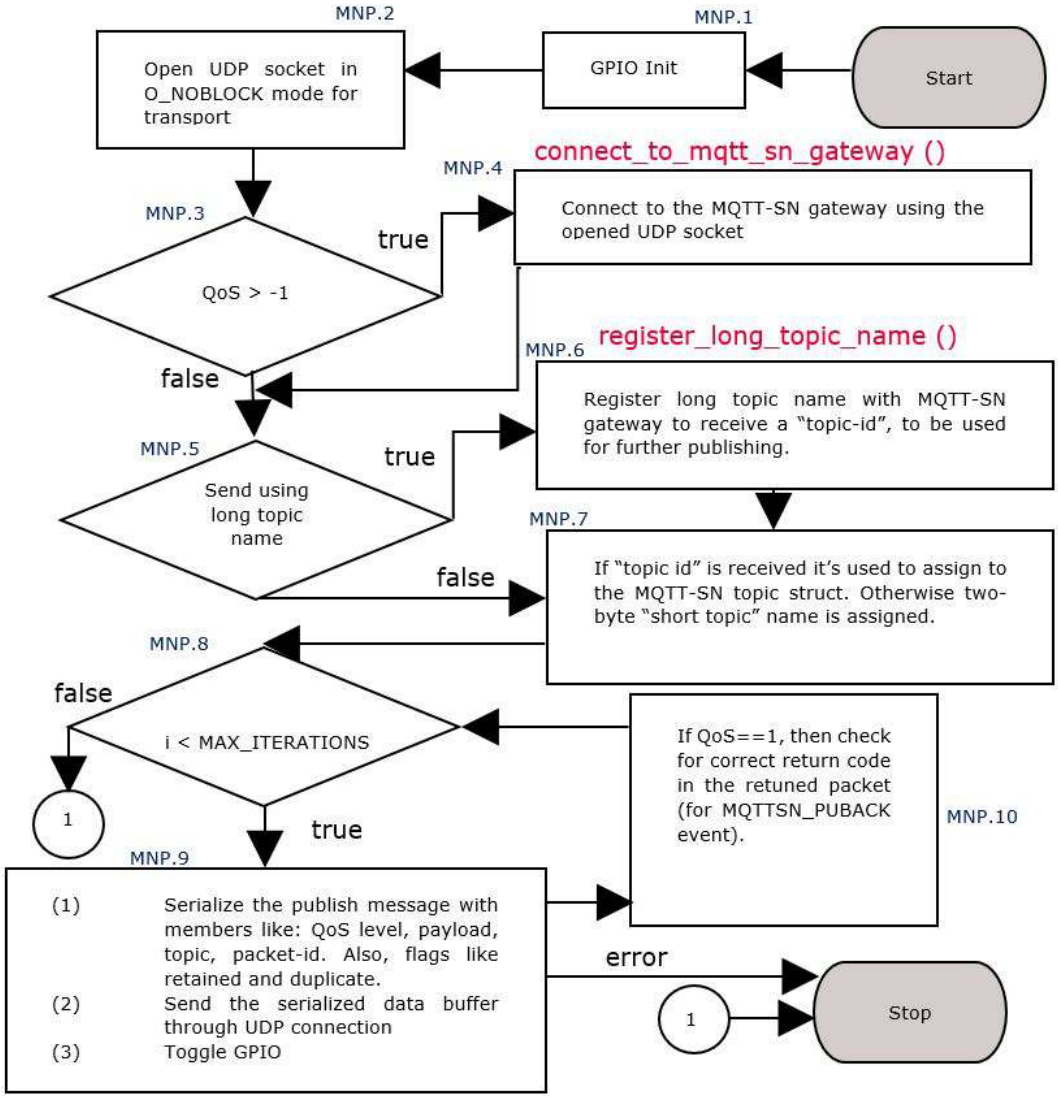


Figure 5.5: MQTT-SN publisher application flowchart on ZephyrRTOS

the gateway through the opened UDP socket. The connection to the gateway can be considered successful if the corresponding deserialised MQTTSN_CONNACK response packet received immediately have a valid return value.

MQTT-SN publish packets can be sent using a size optimised short-topic names (two-byte character) or using a topic-id(integer) received from the gateway (a registered long-topic with the gateway receives the corresponding topic-id). The advantage of using the short-topic for publishing is that, the extra topic registration step with the gateway can be avoided. Registering a long topic-name (register_long_topic_name ()) with the gateway has similar steps as mentioned before for connect_to_mqtt_sn_gateway (),

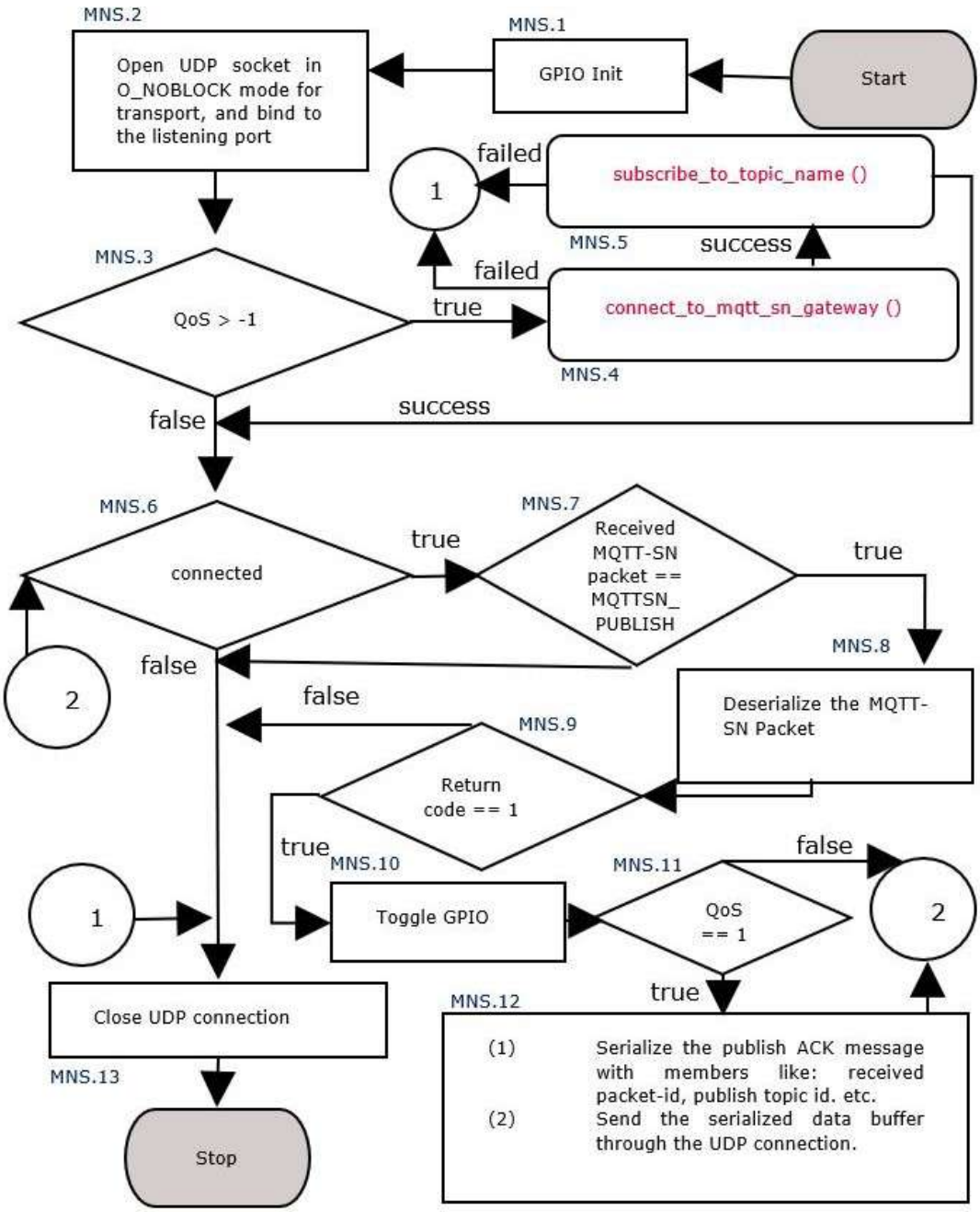


Figure 5.6: MQTT-SN subscriber application flowchart on ZephyrRTOS

but in this case, MQTT-SN specific topic structure is used along with the serialisation api designed specifically for topic registration. These are shown in MNP.5, MNP.6 and MNP.7. The actual sending of 500 MQTT-SN packets is shown with MNP.8, MNP.9 and MNP.10. As mentioned before i'll use a serialised data buffer containing

payload, QoS, topic etc for sending through the UDP socket before toggling the GPIO to mark a successful publish event. For the highest QoS publish (QoS = 1) we can only toggle GPIO, if we receive a correct return code in the deserialised reply packet (MQTTSN_PUBACK) from the gateway (as shown in MNP.10). The socket can be closed after all the transmissions are done.

Figure 5.6 represents the MQTT-SN subscriber flow. As explained previously for the MQTT-SN publisher, MNS.1 to MNS.4 do initialisation till connecting to the gateway if QoS is greater than -1. subscribe_to_topic_name () is called (MNS.5) immediately after connecting to the gateway with a specific topic, so that all the future publish messages on that topic will be received by this node (This step can be skipped in case of multicasted QoS -1 MQTT-SN packets as mentioned before). MNS.6 to MNS.12 depicts the busy loop employed on the subscriber for receiving a MQTT-SN publish packet and toggles its GPIO to signal a successful reception. On receiving an MQTTSN_PUBLISH packet on the corresponding opened UDP socket i'll check for the return value encoded in the deserialised packet before extracting the payload (MNS.7 to MNS.9). Toggling the GPIO and extracting the payload can be done if the return code is valid (MNS.10). Before waiting for another publish message, it's required that a publish acknowledgment MQTT-SN message be sent back (shown in MNS.11 and MNS.12) for higher QoS value (QoS = 1). The UDP socket for MQTT-SN communication can be closed after receiving all the messages.

5.1.1.3 Mixed traffic implementations

Based on the statistics collected from the analysed waveforms generated (using GPIO toggle) from MQTT and MQTT-SN publisher/subscriber implementations, multicasted MQTT-SN with QoS-1 is found to be having minimum jitter (summarised in Section 6.1). The real-time M2M communication needed to be fulfilled is done using the "multicasted MQTT-SN with QoS -1" based on the results. The next step is to ensure that the chosen real-time traffic maintains its benchmark performance (behavior in the absence of any noise/interference) in all conditions.

We deliberately congest the Zephyr network queues with noise packets (either UDP or TCP) in the presence of real-time traffic to observe the resulting performance. Then a socket level priority mapping exposed by the Zephyr BSD socket is exploited to shape the traffic such that, the increased jitter caused by the noise traffic on real-time is reduced to a minimum (the corresponding results are summarised in Section 6.2).

Figure 5.7 represents the proposed mixed traffic implementation (for publishing M2M traffic), which will congest the Zephyr network queue with UDP/TCP noise in

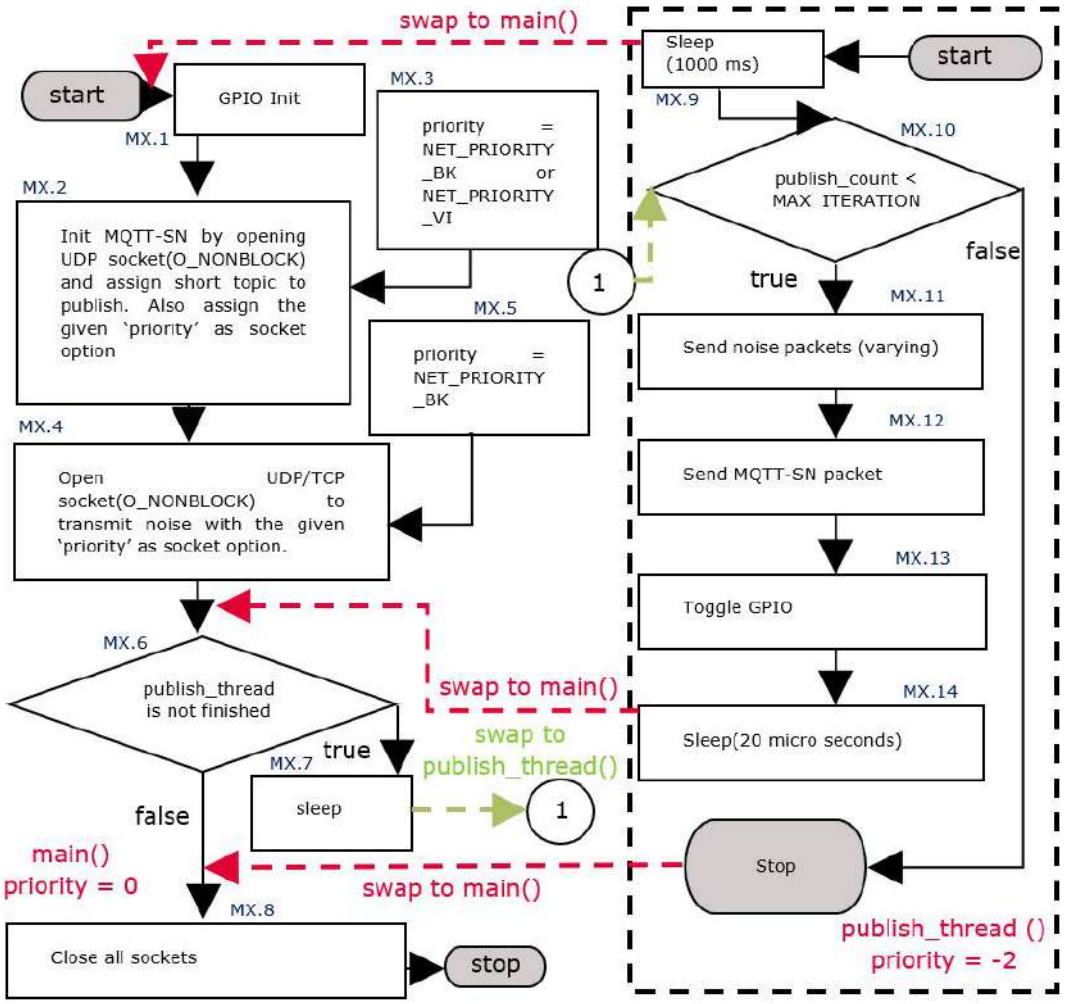


Figure 5.7: Mixed traffic implementation flowchart (containing MQTT-SN QoS-1 publisher and noise traffic) on ZephyrRTOS

the presence of "real-time" multicated MQTT-SN QoS-1 packets. The same MQTT-SN subscriber as mentioned in Figure 5.6 is reused as the subscriber node. The mixed traffic publisher app consists of the *main()* thread (thread priority = 0) and the *publish_thread()* (with thread priority = -2) as shown.

The *main()* thread is responsible to initialising GPIOs (MX.1), opening corresponding real-time and noise sockets, assigning their socket priorities (MX.2 to MX.4), initialising MQTT-SN publish topic structures ,and waiting on a busy loop (MX.6 and MX.7) till the *publish_thread()* thread is completed. The Zephyr system network transmit threads have higher priority than the *main()* thread (meaning their actual numerical priority, -1 is less than the priority of *main()*). The *publish_thread()*

(which is responsible for the actual transmission of real-time and noise packets) needs to be non-preemptable by Zephyr network threads, and hence a priority value of -2 is selected (In effect *publish_thread()* have higher priority than Zephyr network transmit thread).

The first sleep in *publish_thread()* swaps the lower priority main thread to finish the initialisation steps and enter into sleep mode. The Zephyr scheduler will swap the *publish_thread()* during this phase and it'll fill the network queues with real-time and noise traffic, followed by the toggling of GPIO (MX.10 to MX.13) and enter into a brief sleep to allow the Zephyr network transmit thread to service/send the queued packets through the ethernet. As shown in Figure 5.7 the *main()* thread will also finish one iteration (MX.6 and MX.7) during this phase, and again the *publish_thread ()* will be swapped back to continue its trasmit cycle. After all the transmissions are completed the *main()* thread will close the sockets (MX.8) and exit the application.

5.1.1.4 gPTP sample application for time synchronisation

The gPTP sample application provided by the Zephyr project (zephyr/samples/net/gptp/) was mainly reused for querying about the state of time synchronisation. The *ClockTargetPhaseDiscontinuity* application interface standard (IEEE 802.1 AS) implemented in Zephyr as *gptp_register_phase_dis_cb ()* is used in the sample application for registering a callback. This callback is called after a phase discontinuity is sent by the PTP grandmaster. Information about the current grandmaster's identity and its phase change is displayed in this callback. The Zephyr RTOS non-standard gPTP API *gptp_get_port_data ()* is also made use for displaying the current status of the gPTP node after a fixed time (e.g. 15 min). The current status could be *MASTER* to indicate gPTP grandmaster or *SLAVE* to indicate gPTP slave or *FAILED* to indicate invalid gPTP state.

5.1.2 Data/Measurement analytics implementations

In this section the physical measurement setup, the logic performed on the collected pulses (to measure the latency of a packet to reach from publisher to subscriber, and thus overall jitter) and its implementation with R code is explained.

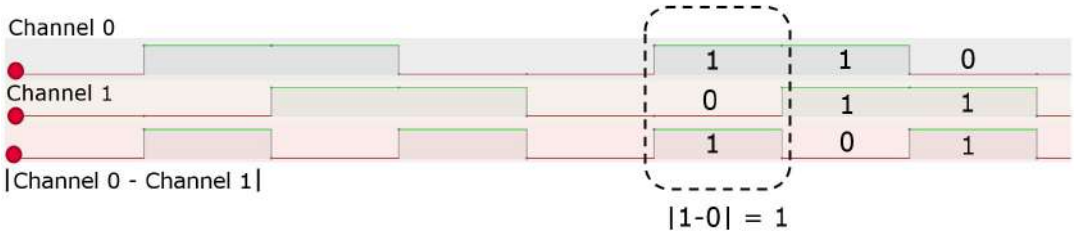


Figure 5.8: GPIO toggled waveforms obtained from publisher (Channel 0) and subscriber (Channel 1), and the corresponding resultant waveform (|Channel 0 - Channel 1|).

5.1.2.1 Measurement logic and waveforms

The main logic to measure the latency it takes for a publish packet to reach the subscriber node (from the publisher) is by measuring the GPIO toggles. The publisher node toggles its GPIO pin as soon as it publishes the packet. On the subscriber side, as soon as the packet is received, its corresponding GPIO pin is toggled. The two physical GPIO pins of the XMC4500 reference board, from both publisher and subscriber is connected to a Saleae logic analyser probe to capture the toggle pulses (low/high) digitally. These digital waveforms are exported as a Comma-Separated Values (CSV) file from the Saleae software for further analysis using R Studio (R programming).

Figure 5.8 represents all the waveforms (*.csv file) opened in 'pulseView' software. 'Channel 0' is the GPIO toggle waveform captured from the publisher node, and 'Channel 1' from the subscriber node. The third waveform represented as ' |Channel 0 - Channel 1| ' is the resulting waveform generated from R-code for statistical analysis. Thus the third waveform is just the modulus of the difference between 'Channel 0' and 'Channel 1'. An example can be seen with the dotted box annotated in the figure. 'Channel 0' toggles its GPIO to high(1) when it publishes, but at that same instant 'Channel 1' has a low (0) signal because it haven't received the packet yet. And hence this pulse width where 'Channel 0' and 'Channel 1' have different logic is accumulated to the overall transmit latency of the publish packet. As soon as the packet is received the 'Channel 1' will toggle its pin to high and thus latency measurement is stopped for that particular transmit packet.

5.1.2.2 R-code implementation for analytics and visualisation

Listing 5.1: R code implementation for generating and visualising measurement analytics

```
1  #code to extract latency from two GPIO pulse waves
2  df <- read.csv(file = "/home/digital.csv", stringsAsFactors = FALSE)
3  df$logic <- ifelse(df$Channel.0-df$Channel.1 != 0 , 1, 0)
4  jitter <- 0
5  for(i  in 1:length(df$logic) ){
6      if(i+1 < length(df$logic) && i != 1 && (df[i, "logic"] == 1 &&
7               df[i + 1, "logic"] == 0))
8      { jitter <- c(jitter, df[i+1, "Time..s."] - df[i, "Time..s."]) }
9  }
10
11 #ggplot is used for histogram plotting
12 df <- data.frame(a = jitter)
13 ggplot(df, aes(x=a)) +
14   geom_histogram(colour="black", fill="white", bins=80) +
15   geom_vline(aes(xintercept=mean(jitter)), color="blue",
16   linetype="dashed", size=1)+ theme_bw(base_size = 22) +
17   labs(x="Jitter/time-delta(s)", y="Ethernet packets")
18
19 # plot() is used for plotting Jitter in Y-axis
20 plot(jitter, type="b" ,pch = 21, xlab = "Ethernet packets",
21   ylab ="Jitter/time-delta(s)", cex.main=2.2, cex.lab=2.2,
22   cex.axis=2.2, cex.sub=2.2, bg = "red", col = "black",
23   cex = 2, lwd = 0.5)
24 abline(h=mean(jitter), col="blue", lwd=4)
```

Listing 5.1 represents the corresponding R code that'll calculate the resulting latency pulse (|Channel 0 - Channel 1|) as shown in the column D of Figure 5.9. Lines 1 to 9, calculate the time delta of the resulting latency pulse (after reading from digital.csv file) as shown in the Column E (Time delta), of Figure 5.9. These generated time-delta between two successive GPIO toggles are then used to plot the jitter histogram with ggplot() (Line 11 to 17). Lines 19 to 24 will plot the jitter on Y-axis, with ethernet packets on X-axis (as shown in Figure 6.5).

5.2 Hardware setup for measurements

Two Infineon XMC4500 based reference boards (*control nodes*) connected to a switch via Ethernet cables are shown in Figure 5.11. Siemens SCALANCE XR-500 Ethernet

	A	B	C	D	E
1	Time [s]	Channel 0	Channel 1	|Channel 0 – Channel 1|	Time delta
2	0	0	0	0	
3	9.894660116	1	0	1	0.000464229999999
4	9.895124346	1	1	0	
5	9.896022368	0	1	1	0.000220828
6	9.896243196	0	0	0	
7	9.897421932	1	0	1	0.000220113999999
8	9.897642046	1	1	0	
9	9.898822552	0	1	1	0.000220042
10	9.899042594	0	0	0	

Figure 5.9: CSV file containing measurement data

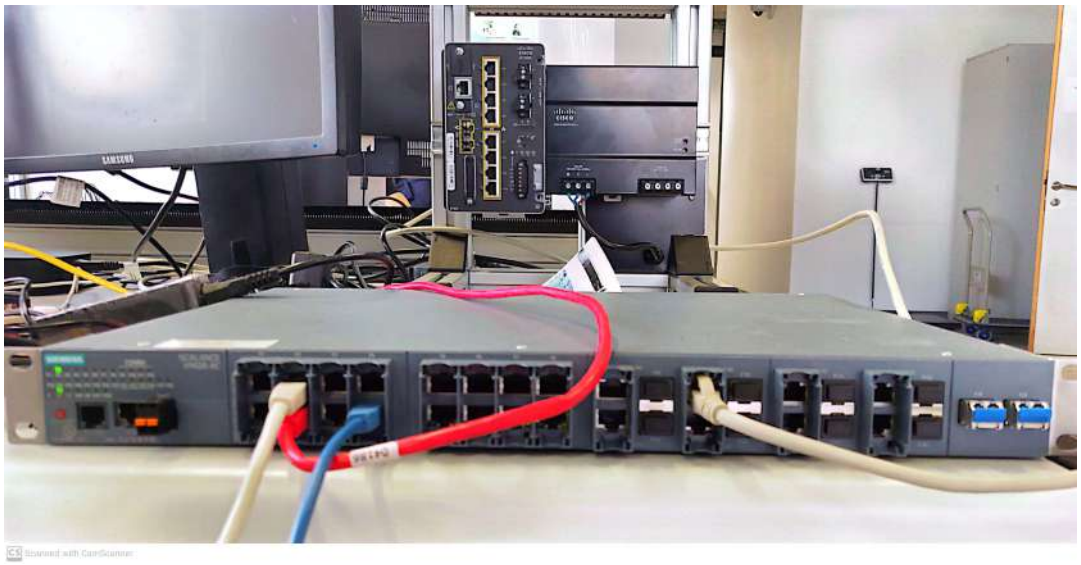


Figure 5.10: Hardware setup containing Siemens Ethernet switch and connections

switch as shown in Figure 5.10 is used to connect all the *control nodes* and the *Host PC*. The *control nodes* are probed by the *Saleae* logic analyser as shown in Figure 5.11 for measuring the latency and jitter from GPIO pins.

The Zephyr RTOS code along with Ethernet driver implementation and the application code is compiled and flashed from two different laptops. The compiled binary is flashed to the *control nodes* via the SEGGER JLINK debugger present in the XCM4500 reference board.

5.3 Software tools for measurements

The software tools used for measurement and M2M communication are described in this section.

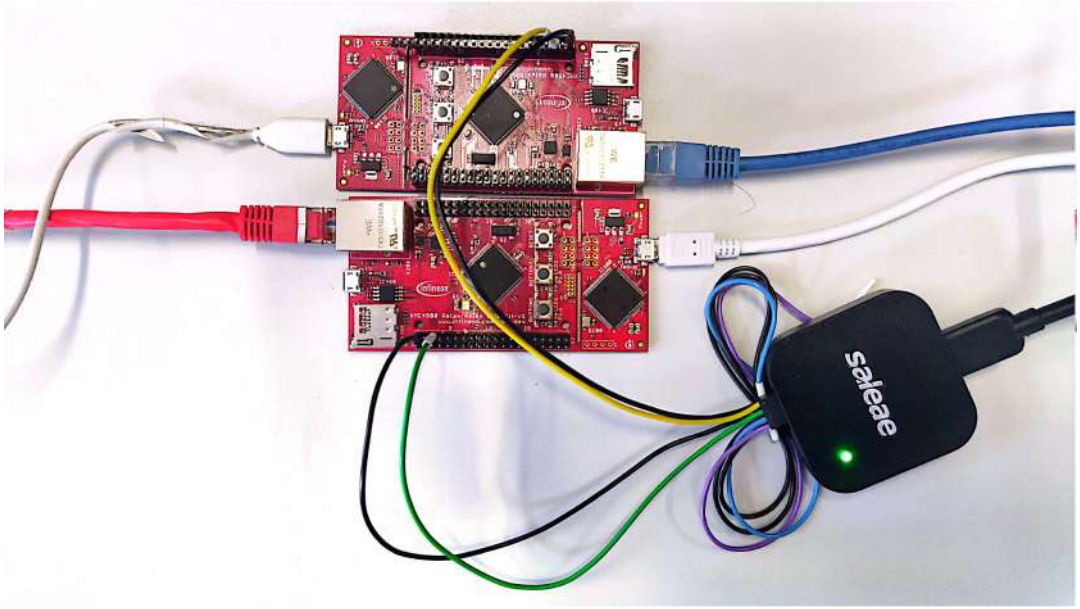


Figure 5.11: Hardware setup containing control nodes and logic analyser

5.3.1 MQTT-SN Gateway and MQTT broker

MQTT-SN connection, subscribe and publish messages coming from a subscribe node (myclientid_subscriber) and a publisher node (myclientid_publisher) with a QoS level of 1 is explained in this section to depict the co-working of an MQTT-SN Gateway and an MQTT broker.

Figure 5.12 represents the MQTT-SN Gateway running on the host Ubuntu machine. The open-sourced Eclipse Paho MQTT-SN Gateway with the config file 'gateway.conf' is started to manage the M2M communication coming from publisher (Zephyr RTOS) and subscriber (Zephyr RTOS). The dashed arrows (<– and –>) represent the incoming and outgoing traffics from the MQTT-SN Gateway, whereas the double dashed arrows (<== and ==>) represent the outgoing and incoming traffics to the MQTT broker (Eclipse mosquitto) running on the same machine.

The corresponding messages received and sent by the MQTT mosquitto broker running on the host machine can be seen in Figure 5.13. The initial 'CONNECT' message is the connection request from the subscriber node (<– myclientid_subscriber), which is then routed to the MQTT broker for connection request by the Gateway (==> myclientid_subscriber). The'CONNACK' reply from the broker on successful connection (<== myclientid_subscriber) is then sent back to the subscriber by the Gateway (–> myclientid_subscriber). Similarly, the 'SUBSCRIBE' and its 'SUBACK'

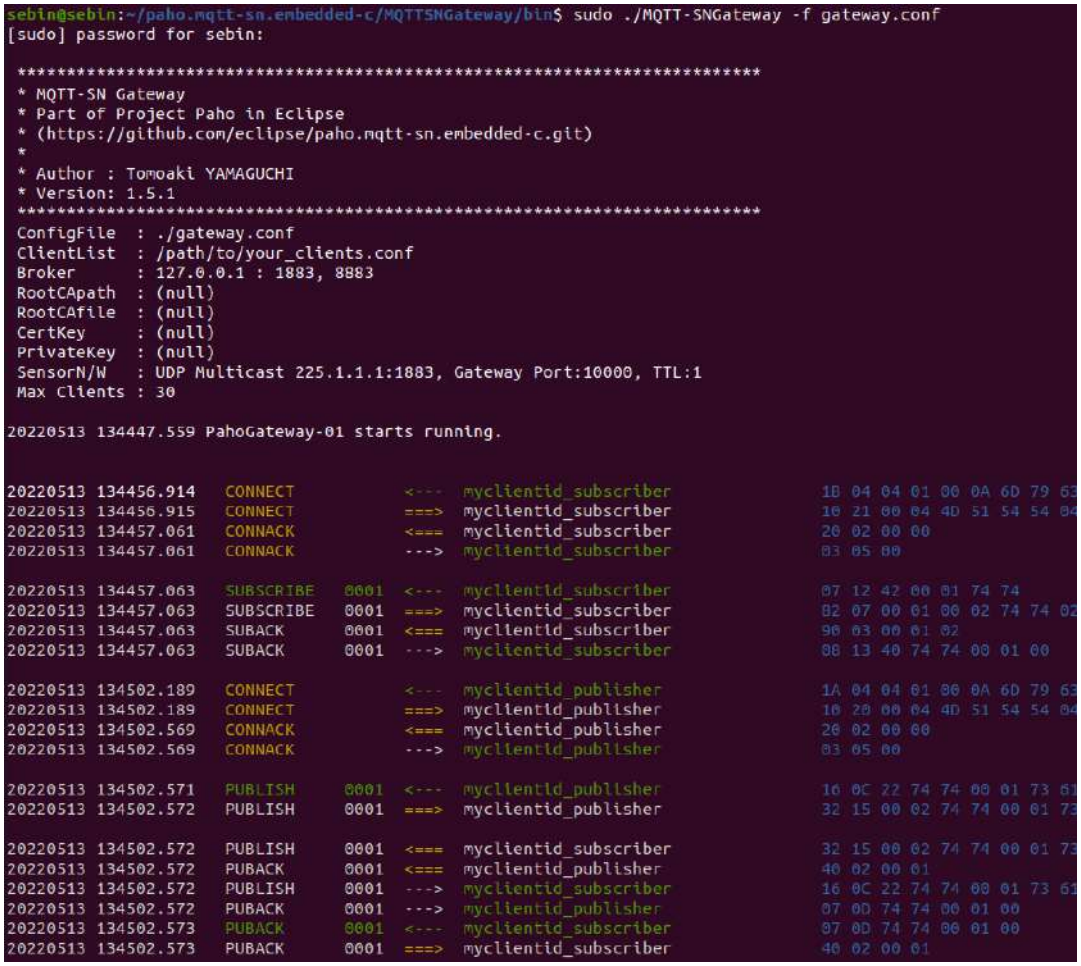


Figure 5.12: Paho MQTT-SN Gateway running on Ubuntu (host) for servicing QoS 1 traffic.

messages for subscribing to a topic from the subscriber goes through the Gateway and broker as shown in Figure 5.12.

The publisher node (myclientid_publisher) goes through a similar connection handshake mechanism as the subscriber. Next, the 'PUBLISH' message received by the Gateway from the publisher (<– myclientid_publisher) is routed to the broker first (==> myclientid_publisher) and then forwarded to the subscriber node (<== myclientid_subscriber and –> myclientid_subscriber). The broker will acknowledge the publish by sending a 'PUBACK' (<== myclientid_publisher) to the publisher. Also, the publish acknowledgment from the subscriber is received by the broker (<– myclientid_subscriber and ==> myclientid_subscriber), so that the MQTT-SN QoS 1 handshake is maintained.

```
sebin@sebin:~$ sudo mosquitto -v
[sudo] password for sebin:
1652442282: mosquitto version 2.0.14 starting
1652442282: Using default config.
1652442282: Starting in local only mode. Connections will only be possible from clients runnir
1652442282: Create a configuration file which defines a listener to allow remote access.
1652442282: For more details see https://mosquitto.org/documentation/authentication-methods/
1652442282: Opening ipv4 listen socket on port 1883.
1652442282: Opening ipv6 listen socket on port 1883.
1652442282: mosquitto version 2.0.14 running

1652442296: New connection from 127.0.0.1:48682 on port 1883.
1652442296: New client connected from 127.0.0.1:48682 as myclientid_subscriber (p2, c1, k10).
1652442296: No will message specified.
1652442296: Sending CONNACK to myclientid_subscriber (0, 0)
1652442297: Received SUBSCRIBE from myclientid_subscriber
1652442297:     tt (QoS 2)
1652442297: myclientid_subscriber 2 tt
1652442297: Sending SUBACK to myclientid_subscriber
1652442302: New connection from 127.0.0.1:48684 on port 1883.
1652442302: New client connected from 127.0.0.1:48684 as myclientid_publisher (p2, c1, k10).
1652442302: No will message specified.
1652442302: Sending CONNACK to myclientid_publisher (0, 0)
1652442302: Received PUBLISH from myclientid_publisher (d0, q1, r0, m1, 'tt', ... (15 bytes))
1652442302: Sending PUBLISH to myclientid_subscriber (d0, q1, r0, m1, 'tt', ... (15 bytes))
1652442302: Sending PUBACK to myclientid_publisher (m1, rc0)
1652442302: Received PUBACK from myclientid_subscriber (Mid: 1, RC:0)
1652442302: Received PUBLISH from myclientid_publisher (d0, q1, r0, m2, 'tt', ... (15 bytes))
1652442302: Sending PUBLISH to myclientid_subscriber (d0, q1, r0, m2, 'tt', ... (15 bytes))
1652442302: Sending PUBACK to myclientid_publisher (m2, rc0)
1652442302: Received PUBACK from myclientid_subscriber (Mid: 2, RC:0)
```

Figure 5.13: MQTT broker (mosquitto) running on Ubuntu (host) for receiving and sending traffic to Paho MQTT-SN Gateway.

Listing 5.2: mosquitto(MQTT) and Paho MQTT-SN commands

```
1 # Run mosquitto server:
2 sebin@ubuntu:~$ sudo mosquitto -v -c /etc/mosquitto/mosquitto.conf
3
4 # Run mosquitto subscriber:
5 sebin@ubuntu:~$ mosquitto_sub -t 'sensors' -v
6
7 # Run MQTT-SN Gateway
8 sebin@ubuntu:~$ sudo ./MQTT-SNGateway -f gateway.conf
```

Listing 5.2 show the basic commands that are used for MQTT and MQTT-SN communication from the host side (Ubuntu). Line 2, starts a MQTT broker (Eclipse mosquitto) giving mosquitto.conf as the configuration file. For accepting direct TCP based MQTT communication over ethernet from other nodes (non-local), the configuration file needs to be modified to add 'listner 1883 0.0.0.0', 'bind_interface enp4s0' and 'allow_anonymous true' (emp4s0 is the Ubuntu Ethernet interface), so that the broker will listen for non-local communication over 1883 port. Line 5, is the

command used to start a local MQTT subscriber(mosquitto_sub) to listen to the topic 'sensors'. Line 8 can be used to start the compiled MQTT-SN Gateway (Eclipse Paho) binary, giving the file 'gateway.conf' as the configuration file argument. The default gateway.conf 'BrokerName' field is also modified into '127.0.0.1' (localhost) for the gateway to directly communicate with the MQTT broker running on the same host machine.

5.3.2 Tools for network, schedule and signal analysis

5.3.2.1 Traceanalyzer

The Zephyr RTOS trace analyser tool *Traceanalyzer* by *Percepio* was used to get a visual timeline of different tasks running on Zephyr RTOS. *Traceanalyzer* supports streaming mode over Real Time Transfer (RTT), which helps to get real-time data about the running tasks with the help of the SWD interface of the SEGGER JLINK debugger.

Figure 5.14 represents the visual trace where the application thread *thread0_id* is first sending Ethernet packets over a socket. The high priority Zephyr network transmit thread *tx_q[1]* services the incoming packets first followed by the low-priority thread *tx_q[0]*. During the transmission of thread *tx_q[0]* an interrupt is raised by *Zephyr ISR* thread because of an incoming Ethernet packet. The Ethernet driver receive thread *xmc_eth_rx* services the incoming packet after *tx_q[0]* has finished.

5.3.2.2 Wireshark

The free and open-source network protocol analyser *Wireshark* is used to monitor and troubleshoot the Ethernet packets transmitting between XMC4500 *control nodes* and the *Host PC*. *Wireshark* is run on the Ubuntu based *Host PC* on its Ethernet network interface. The linuxPTP daemon (*ptp4l*) is also running on the same Ethernet port for time synchronising between the XCM4500 and Ubuntu. Figure 5.15 represents the gPTP Ethernet packets captured while troubleshooting the implemented PTP driver in Zephyr (as mentioned in Section 4.2.1).

The information about the current synchronisation status of the gPTP grandmaster (Infineon_45:00:*) is sent over the *announce* message as shown in Figure 5.15. gPTP messages like *Peer_Delay_Req_Message*, *Peer_Delay_Resp_Message*, *Peer_Delay_Resp_Follow_up* etc. are also sent to calculate the propagation delay between XMC4500 and the linuxPTP daemon in Ubuntu.

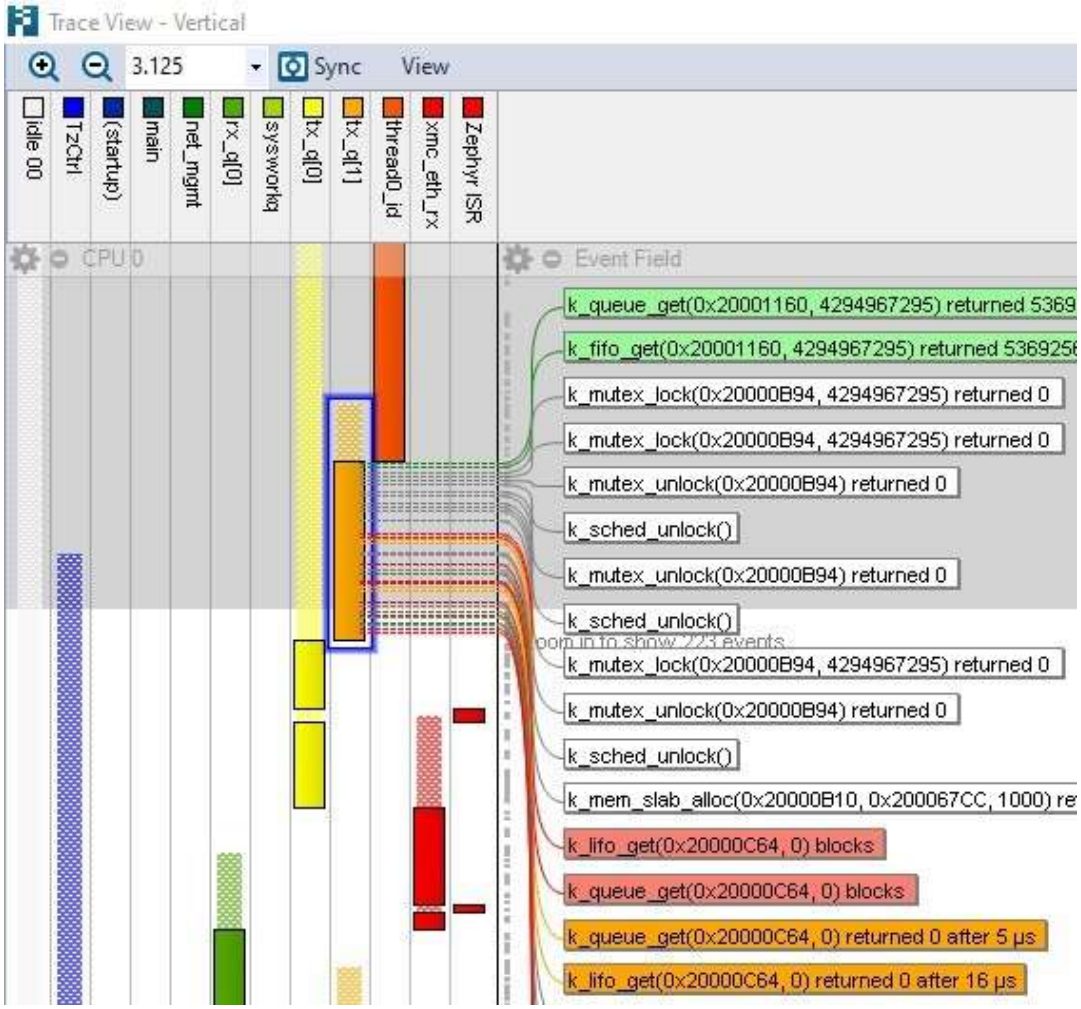


Figure 5.14: Real-time ZephyrRTOS application trace/schedule captured with Trace-analyzer

5.3.2.3 Saleae logic analyser

The logic analyser tool from *Saleae* was used for recording the GPIO toggle pulses from two XMC4500 *control nodes*. The GUI provided by *Saleae* is shown in Figure 5.16. The collected pulse waveform can be later exported as a CSV file for measuring the latency and jitter as explained in Section 5.1.2.1.

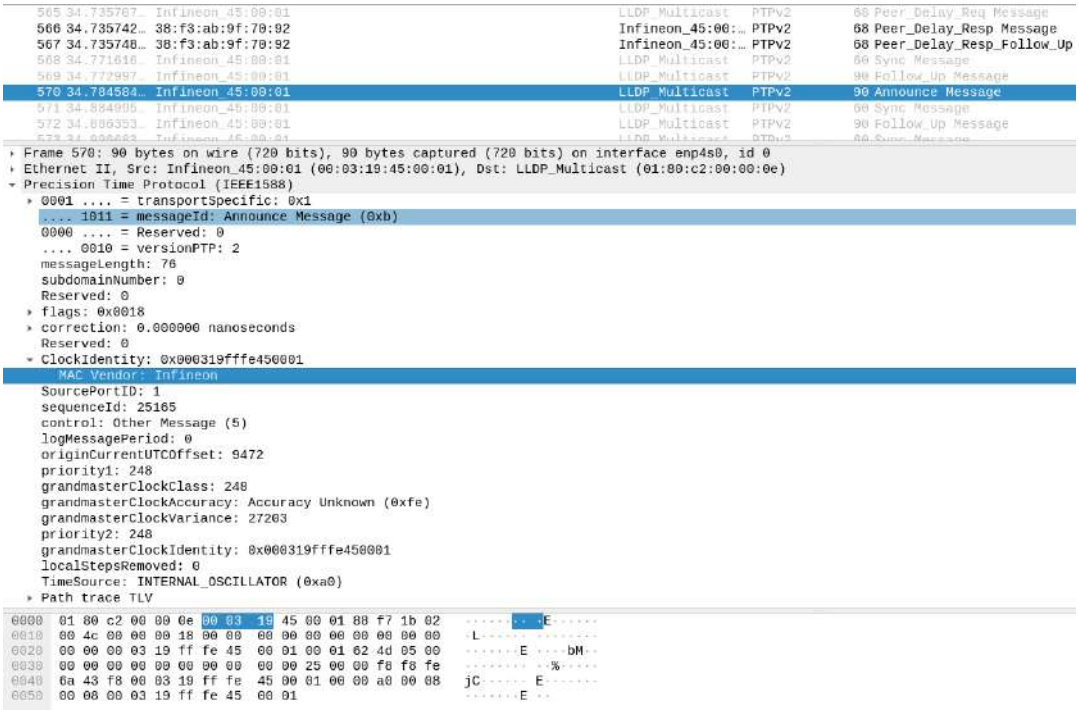


Figure 5.15: gPTP Ethernet packets captured with wireshark

Figure 5.16: GPIO toggles captured in Saleae logic analyser

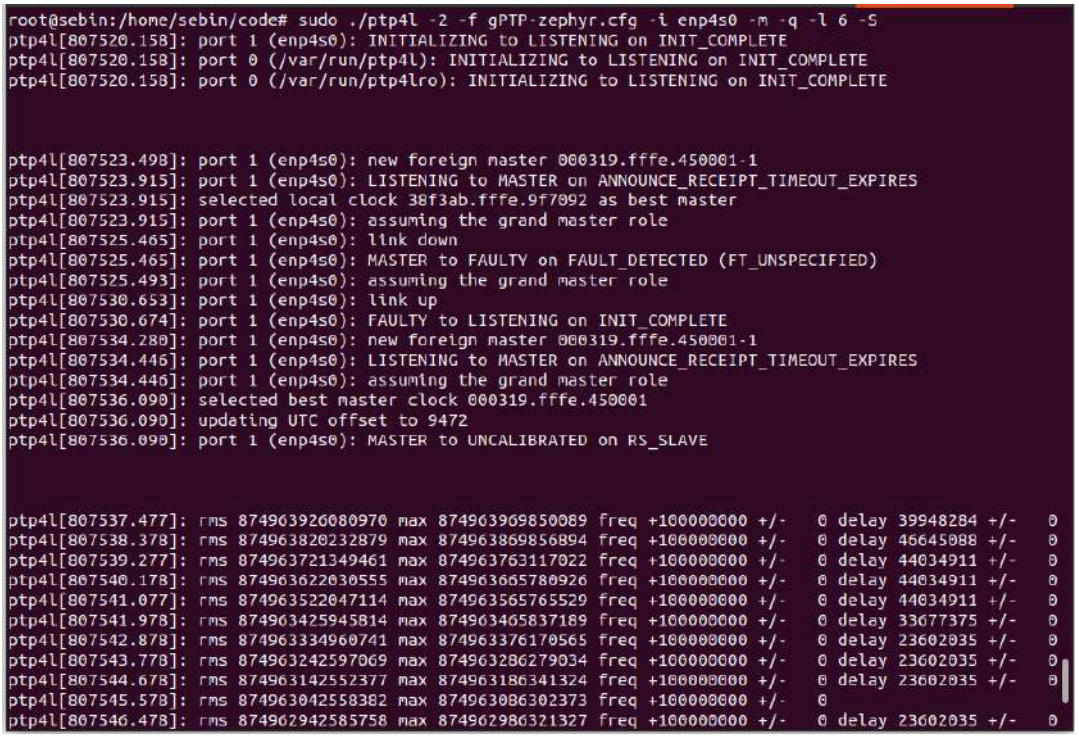


Figure 5.17: gPTP time synchronisation logs from ptp4l

5.3.3 PTP deamon in Linux

The *LinuxPTP* project [26] implements PTP according to the IEEE 1588 standard for Linux based OS. *LinuxPTP* supports both hardware and software timestamping along with support for the Linux PTP Hardware clock (PHC) subsystem. PTP clocks like Boundary Clock (BC), Ordinary Clock (OC), Transparent Clock (TC) etc. are also supported in all major network transports (TCP/UDP/raw Ethernet).

The implementation package includes *ptp4l* and *phc2sys* programs for clock synchronisation. The Ubuntu based *Host PC* which was used for testing time synchronisation was not equipped with hardware timestamping. Hence i've only used the *ptp4l* daemon which synchronises Ubuntu system clock with the attached master clock (*control node*). Figure 5.17 shows the synchronisation logs from the *ptp4l* daemon running on Ubuntu, when the XMC4500 with the PTP driver is implemented in Zephyr RTOS. The Ubuntu shell command used to launch the daemon is shown in Listing 5.3.

5.3.4 Network setup tools in Linux

Listing 5.3: Linux PTP4L(daemon) command

```
1 # Run ptp4l daemon of linuxptp project
2 sebin@ubuntu:~$ sudo ./ptp4l -2 -f gPTP-zephyr.cfg -i enp4s0 -m -q -l 6 -S
```

Listing 5.4: Linux commands for ethernet network interface setup

```
1  # Add static IP address for Ethernet interface in Ubuntu.
2  sebin@ubuntu:~$ ip addr add 192.0.2.2 dev brd + enp4s0
3
4  # Add route so that ARP reply is given from Ubuntu on
5  # seeing ARP  request messages from Zephyr.
6  sebin@ubuntu:~$ ip route add 192.0.2.0/24 dev enp4s0
7
8  sebin@ubuntu:~$ ip addr flush dev enp4s0
9  # Add the MQTT-SN multicast address to the route
10 sebin@ubuntu:~$ ip route add append 225.1.1.1 dev enp4s0
11 sebin@ubuntu:~$ route
12 sebin@ubuntu:~$ ifconfig
13
14 # start UDP server in Ubuntu for listening, using netcat
15 sebin@ubuntu:~$ nc -u -l <port-no>
16
17 # start TCP server in Ubuntu for listening, using netcat
18 sebin@ubuntu:~$ nc -l <port-no>
```

Listing 5.4 lists the basic Linux command line utilities used for network setup. Line 2 adds a static IP address to the ethernet interface (enp4s0), and Line 6 will use the first 24 of that IP address as the network mask. This will ensure a network route is added for traffic coming from the same network address (192.0.2.*), and hence the host (Ubuntu) will be able to communicate (eg: giving Address Resolution Protocol (ARP) replies back). Line 8 will clear the existing IP address information from the interface and Line 10 will append the multicast address '255.1.1.1' used by the MQTT-SN Gateway for node discovery broadcasts. Commands on Line 11 and 12 can be used to view the routing table and ethernet device interface information. The netcat utility commands displayed on Lines 15 and 18 can be used to start servers and listen to incoming UDP and TCP traffics respectively on port '<port-no>'.

6

Results and Analysis

This chapter deals with the analysis of the results obtained from the actual implementation on Zephyr RTOS based on the 'Experimental study'. First, a comparison of performance evaluation of MQTT and MQTT-SN traffic is done in Section 6.1. In Section 6.2 results collected for the mixed traffic approach (hard real-time and noise data/soft-real-time) are compared with regard to its deterministic behavior. Finally the gPTP time synchronisation results are analysed in Section 6.3.

6.1 MQTT and MQTT-SN performance (jitter) measurement

The performance of MQTT is measured in terms of the jitter spread observed with regard to the toggling of GPIO pins for 500 publish packets from publisher to subscriber (as mentioned in Figure 5.2 and Figure 5.3 of 'Experimental study').

The jitter/time-delta histogram spread of MQTT with QoS0 (lowest) and QoS2 (highest) is plotted in Figure 6.3. Figure 6.1 plots the jitter of MQTT with QoS0, with its mean marked with the blue line. It's evident that with the increased QoS, the amount of TCP traffic increases for MQTT, and hence the mean latency for a successful publish packet to reach increased from 1.30 ms to 3 ms (summarized in Table 6.1). It's interesting to observe that its corresponding jitter increased by 80 % with the quality as observed in Table 6.1.

Similarly, the histogram plots for MQTT-SN with QoS-1 (corresponding jitter plot in Figure 6.2) and QoS1 can be compared from Figure 6.4. As seen with MQTT, with the increased QoS, the mean latency increased from 270 µs to 1440 µs. However, a drastic 1144 times increased jitter is observed with the increased quality.

The "multicasted fire-and-forget" approach used for QoS-1 packets contributed towards a very low jitter. In the case of QoS-1 (fire-and-forget), it doesn't require

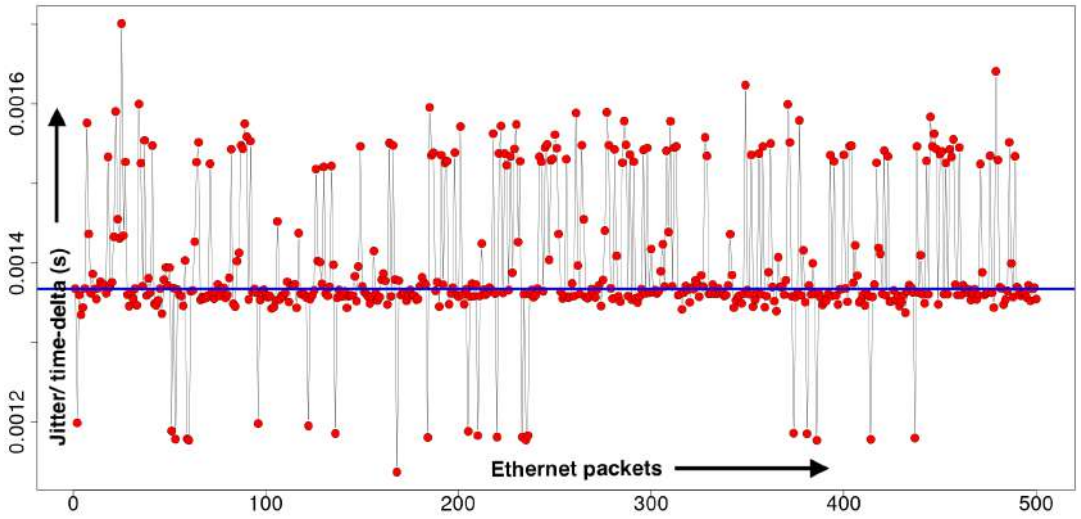


Figure 6.1: Jitter/time-delta spread of 500 MQTT packets with QoS 0

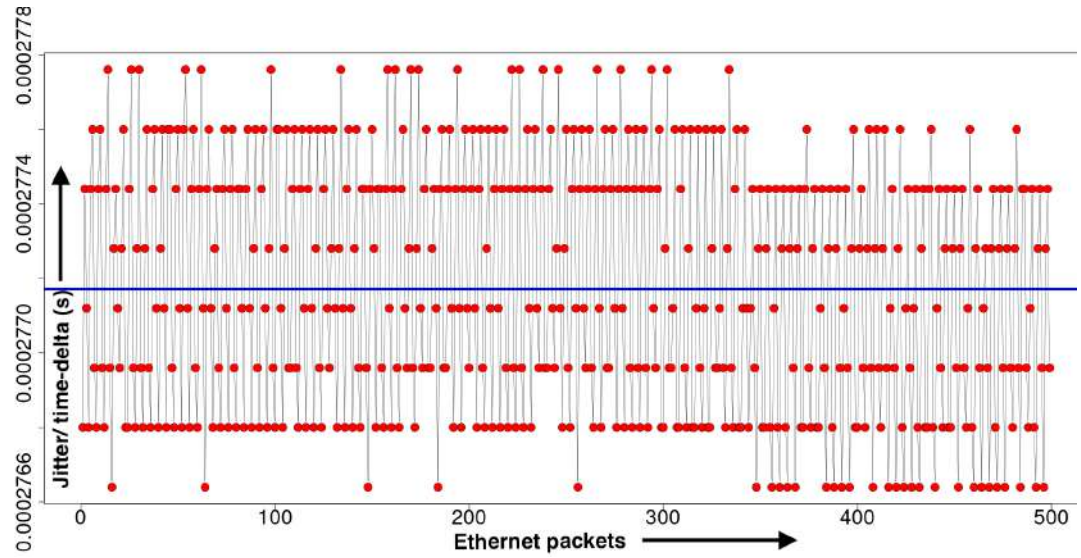
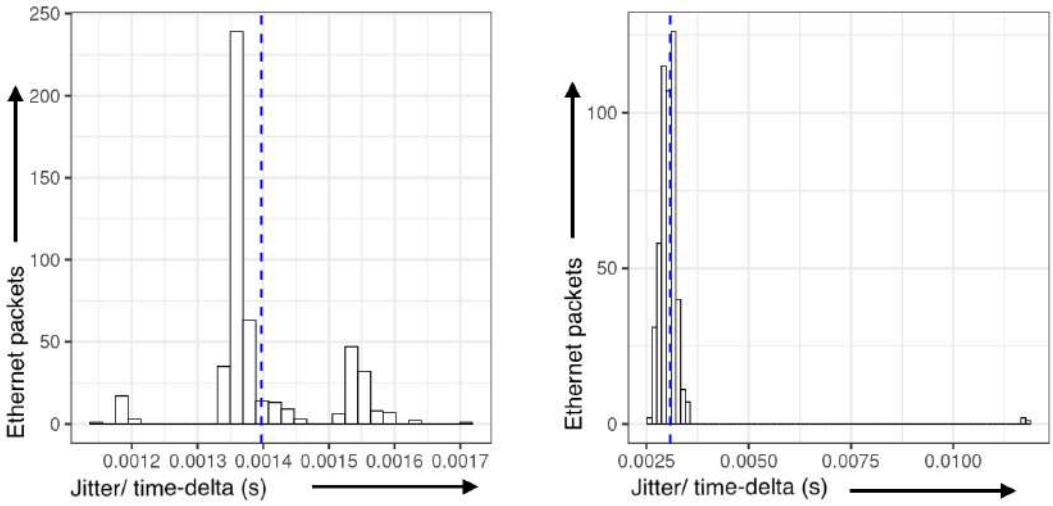


Figure 6.2: Jitter/time-delta spread of 500 "multiplexed MQTT-SN packets" with QoS -1

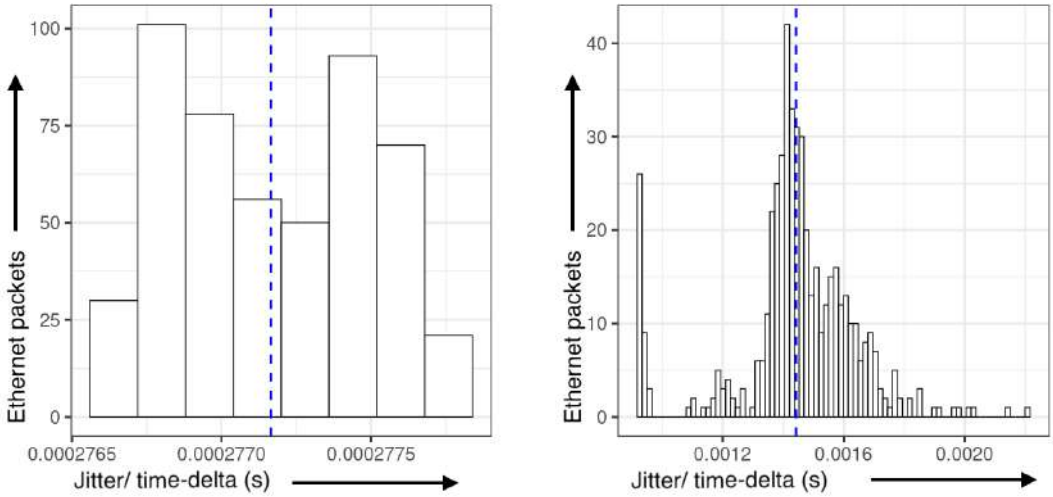
that the MQTT-SN gateway to be present at all between publisher and subscriber to forward the packet. Hence i've directly multicasted the UDP publish packet at the switch level (multicast group based on the topic subscription), resulting in the lowest jitter of 1.12 µs. On the other hand, even though UDP based, MQTT-SN with QoS1 has the highest jitter compared to TCP-based MQTT. MQTT-SN with QoS1 requires that the UDP publish packets be first received by the MQTT-SN gateway,



(a) Jitter/time-delta histogram of 500 MQTT packets with QoS 0

(b) Jitter/time-delta histogram of 500 MQTT packets with QoS 2

Figure 6.3: MQTT histograms of QoS 0 and QoS 2 packets



(a) Jitter/time-delta histogram of 500 "multiplexed MQTT-SN packets" with QoS -1

(b) Jitter/time-delta histogram of 500 MQTT-SN packets with QoS 1

Figure 6.4: MQTT-SN histograms of "multiplexed QoS -1" and QoS 1 packets

re-transmitted to-and-fro (TCP communication) between the MQTT broker (residing in the host) and republished as a UDP packet to the corresponding subscribers.

Table 6.1 can be summarised as: The multicasted MQTT-SN with QoS-1 (multicasted fire-and-forget) achieves the lowest ever jitter compared to all other cases with

Table 6.1: Latency and jitter experinced by 500 MQTT and MQTT-SN publish packets to reach subscriber

Quantity	A	B	C	D
Min.	0.00113728	0.00253024	0.00027664	0.00091840
Max.	0.00170048	0.00354672	0.00027776	0.00220064
Jitter.	0.0005632	0.00101648	0.00000112	0.00128224
1st Qu.	0.001358	0.002895	0.0002768	0.0013817
Median.	0.001365	0.003030	0.0002771	0.0014418
Mean.	0.001397	0.003026	0.0002772	0.0014417
3rd Qu.	0.001412	0.003137	0.0002774	0.0015587

A - MQTT with QoS 0, B - MQTT with QoS 2, C - Multiplexed MQTT-SN with QoS -1, D - MQTT-SN with QoS 1.

different QoS levels. It also reduces the jitter by approximately 503 times compared to the lowest MQTT quality (QoS0). This points to the fact that using the UDP based "multicasted fire-and-forget" MQTT-SN packets for real-time/deterministic communication is best suitable compared to other M2M services like MQTT and other higher MQTT-SN QoS levels.

6.2 Results on deterministic communication

Based on the results from the previous section i've chosen the multicasted MQTT-SN with QoS-1 as the medium/protocol for deterministic/real-time transport. In this section, i'll analyse the performance of the real-time traffic when an additional noise traffic (UDP/TCP) is mixed during transmission and the corresponding improvement observed when a socket-level priority is introduced (as mentioned in Figure 5.7 of 'Experimental study').

6.2.1 Performance of real-time traffic (MQTT-SN with QoS -1) in the presence of UDP based noise congestion

The jitter experienced in the real-time traffic when it's mixed with a random UDP noise is captured in Figure 6.5. The random noise with the same priority (socket level SO_PRIORITY as NET_PRIORITY_BK) as the real-time traffic (SO_PRIORITY as NET_PRIORITY_BK) is modeled to randomly fill the Zephyr transmit networking queue (from minimum to maximum capacity). The corresponding histogram

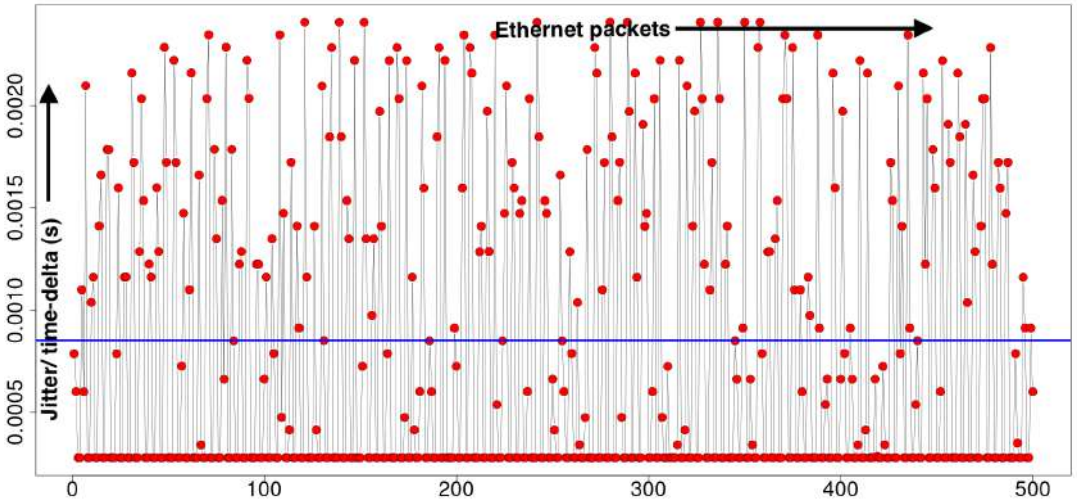
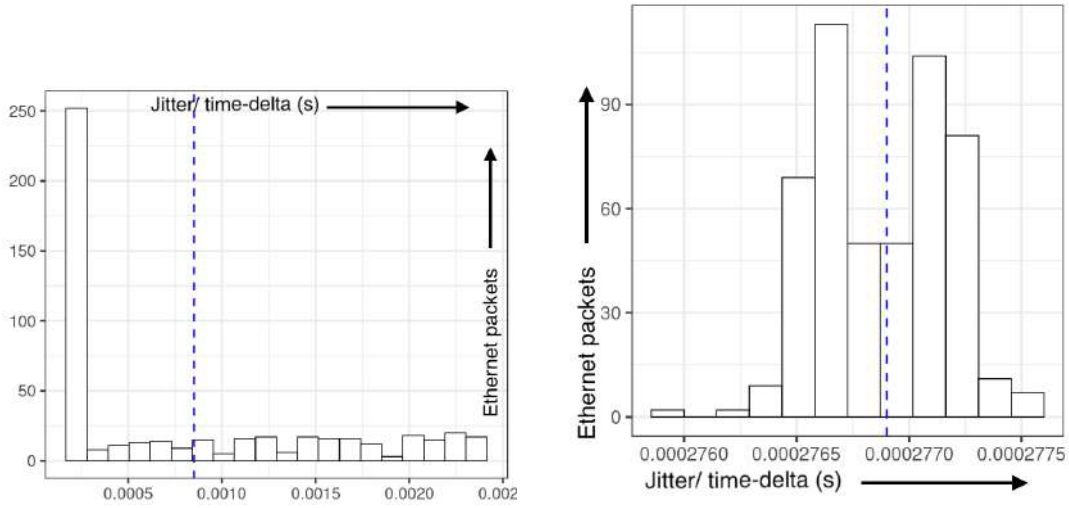


Figure 6.5: Jitter/time-delta spread of 500 "multiplexed MQTT-SN packets" with QoS -1 (real-time packets), in the presence of UDP congestion (with same packet priority)

(Figure 6.6a) depicts the distribution of jitter packets shown in Figure Figure 6.5. The resulting latency ranges from 270 µs to 2400 µs with a jitter of 0.00213 (Table 6.2, column A). An increase in jitter by 1905 times compared to the previous result of multicasted fire-and-forget(Table 6.1, column C) has been introduced due to the congestion introduced in the transmit queues (noise).

Giving higher priority to the real-time traffic (SO_PRIORITY as NET_PRIORITY_VI) from the socket level, forces the Zephyr network stack to enqueue real-time packets into separate higher priority network queues (according to the IEEE 802.1Q mapping described in Section 2.4.3.5). The higher priority "dequeue/transmit thread" (as shown in Figure 4.1, which will hand over the data to the ethernet driver xmc_eth_tx()) associated with this queue is scheduled immediately to send this high priority traffic first, once the application fills up all the network queues with all the packets. It's interesting to note that the NET_PRIORITY_VI packest are first dequeued even before the early arrived noise packets (NET_PRIORITY_BK). The resulting jitter distribution resembles the previous "multicasted fire-and-forget" histogram (Figure 6.4a) is shown in Figure Figure 6.6b. The numerical values corresponding to jitter/range/mean etc of this traffic from Table 6.2 column B, is approaching very near the real-time traffic without noise (Table 6.1, column C). This is a clear indication that the introduced queuing priority mitigates the huge jitter degradation on the noise-mixed "multicasted fire-and-forget" packets (real-time).



(a) Jitter/time-delta histogram of 500 "multiplexed MQTT-SN packets" with QoS -1, in the presence of UDP congestion (with same priority)

(b) Jitter/time-delta histogram of 500 "multiplexed MQTT-SN packets" with QoS -1, in the presence of UDP congestion (with different priority)

Figure 6.6: Jitter/time-delta histograms of 500 "multiplexed MQTT-SN packets" with QoS -1, in the presence of UDP congestion (with and without different packet priorities)

6.2.2 Performance of real-time traffic (MQTT-SN with QoS -1) in the presence of TCP based noise congestion

In this section, the results obtained from mixing a TCP noise instead of UDP with the real-time traffic are listed. An attempt to capture the results having both noise and real-time packets with the same priority(NET_PRIORITY_BK) was not successful. This was because the logic we used to measure the jitter between two consecutive GPIO toggles (in publisher and subscriber) couldn't be performed on the resulting pulse signal. Since some of the UDP MQTT-SN QoS-1 (real-time) packets were lost during transmission and/or because of the huge congestion introduced by the TCP noise (actual noise increases multifold to fulfill the TCP handshake protocol) left us with irregular pulses having huge delay factors between two consecutive toggles. From this it's clear that the UDP MQTT-SN QoS-1 packets are insufficient for real-time transmission along with TCP noise, without any traffic shaping/priority introduced. Concepts like fault tolerance should be considered here for both traffics to co-exist, in a predictable manner.

Figure 6.7 is the jitter displayed by TCP noise mixed with real-time traffic, when priority is introduced. Here UDP based MQTT-SN QoS-1 packets (real-time) are

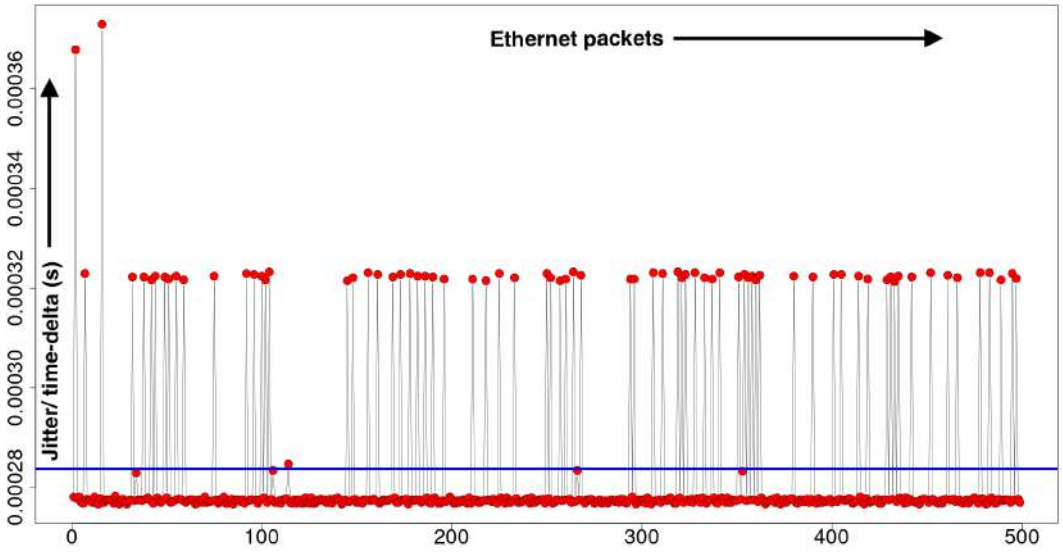
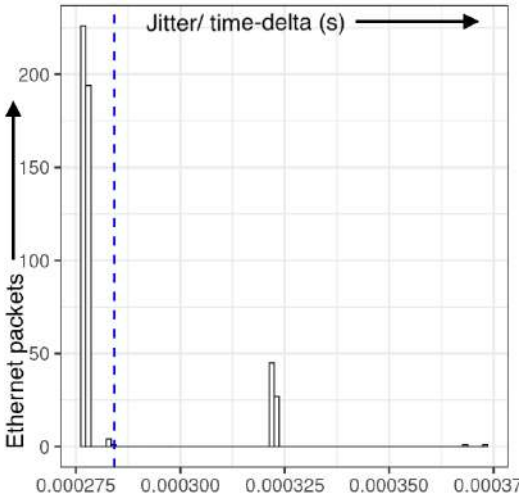


Figure 6.7: Jitter/time-delta spread of 500 "multiplexed MQTT-SN packets" with QoS -1 (real-time packets), in the presence of TCP congestion (with different packet priorities)

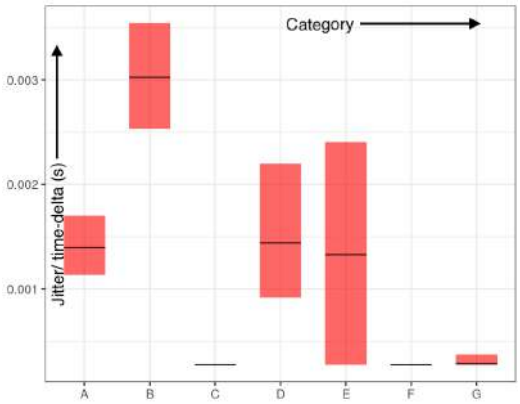
given higher priority (NET_PRIORITY_VI) than the TCP noise (with priority NET_PRIORITY_BK). From its corresponding histogram distribution (Figure 6.8a) it can be seen that a small portion of packets (80 packets out of a total of 500 packets) lie at the extreme right of the histogram graph (latency 325 µs). It's because those real-time packets were colliding with the TCP handshake ACK packets when transmitted. Even though the TCP noise has lower priority, the ACK packets needed to complete the previously send TCP noise was colliding with the real-time traffic. This caused the unusual high latency for these small portions of traffic and hence comparatively high jitter (as seen in Table 6.2, column C). When these 80 high latency packets(outliers) are excluded, the same performance(minimum jitter) as seen with the UDP-based noise (shown in Table 6.2, column D) is obtained.

Based on the data from Table 6.2 it's clear that introducing socket level priority for network queues (transmit queues) brings down the jitter of real-time packets to a minimum (the jitter of Column A is reduced in Column B). Also, even though the same priority based transmission is used along with the TCP noise, the jitter observed (Column C) is higher because of the collision with the ACK packets. On excluding that 16 % of packets with high latency (which was colliding with the ACK), we get a similar performance with the minimum jitter (Column D).

Summary of the all the obtained results are visualised into a spread diagram in Figure 6.8b for better understanding. The jitter measured for each communication



(a) Jitter/time-delta histogram of 500 "multiplexed MQTT-SN packets" with QoS -1 (real-time packets), in the presence of TCP congestion (with different priority)



(b) Final spread comparison (jitter) of different M2M protocols

Figure 6.8: Jitter histogram of real-time packets in presence of TCP noise and final spread (jitter) comparison

Table 6.2: Latency and jitter experinced by 500 real-time publish packets to reach subscriber (in the presence of UDP/TCP noise)

Quantity	A	B	C	D
Min.	0.00027600	0.0002760	0.00027664	0.00027664
Max.	0.00240992	0.0002776	0.00037296	0.00027824
Jitter.	0.00213392	0.0000016	0.00009632	0.0000016
1st Qu.	0.0002765	0.0002766	0.0002770	0.0002770
Median.	0.0002771	0.0002770	0.0002774	0.0002773
Mean.	0.0008516	0.0002769	0.0002842	0.0002773
3rd Qu.	0.0014121	0.0002771	0.0002778	0.0002774

A - MQTT-SN_QoS-1_UDP noise, B - MQTT-SN_QoS-1_UDP noise with priority, C - MQTT-SN_QoS-1_TCP with priority, D - MQTT-SN_QoS-1_TCP with priority (without outliers)

category (A to G) is represented as the red bar (spread), and their corresponding mean jitter value with a black line across the bar. Among MQTT QoS0 (Figure 6.8b, Category A) and QoS2 (Figure 6.8b, Category B), it can be seen visually that MQTT QoS2 has a higher spread (jitter) with a high mean latency. As explained before MQTT-SN with QoS 1 (Figure 6.8b, Category D) has an even large jitter spread

than MQTT. The maximum performance with minimum latency and jitter spread is achieved in the case of multicasted MQTT-SN with QoS -1 (Figure 6.8b, Category C). As mentioned before, this makes multicasted MQTT-SN with QoS -1 the preferred M2M standard for real-time communication (benchmark). When high UDP-based congestion is added along with the transmission of real-time traffic, a large jitter spread is observed(Figure 6.8b, Category E). The classification of traffic based on priority helps achieve the benchmark performance (Figure 6.8b, Category F), by preempting the noise packets. As observed before the TCP-based noise congestion along with priority mapping has an even higher jitter spread than UDP-based (Figure 6.8b, Category G).

6.3 PTP accuracy measurement

The functionality and correct working of the XMC4500 PTP driver implemented for Zephyr RTOS (as described in Section 4.2.1) was initially tested using the *linuxPTP* daemon. XMC4500 with the PTP enabled was directly connected to an Ubuntu machine running *ptp4l* and the PTP control messages were analysed using *wireshark*. It was observed that the PTP synchronisation messages were correctly trasferred to-and-from Zephyr gPTP stack with the help of hardware timestamp enabled PTP driver. The captured PTP network packets during time synchronisation is shown in Figure 5.15. Section 5.3.3 shows the corresponding synchronisation logs from *ptp4l*. The Zephyr gPTP sample application (Section 5.1.1.4) gave information about different states assigned to the gPTP stack during synchronisation.

The accuracy of implemented PTP driver for XMC4500 along with the Zephyr gPTP stack was measured using the same logic explained in Section 5.1.2. Two different Zephyr gPTP stack enabled XMC4500 were connected back-to-back using Ethernet for taking accuracy measurement. Each device will toggle its GPIO pin when a PTP alarm expires every other second (also know as "1 Pulse Per Second (PPS) measurement") as mentioned in Section 4.2.1.1.

Although Zephyr gPTP stack offeres an accuracy in the range of nanoseconds, the measured accuracy was around 500 ms initially. Then the time drifted away gradually from each other between each XMC4500. The observed sub-optimal result suggests further improvement of the PTP driver by carefully analysing PTP packets from the Zephyr gPTP stack.

7

Conclusion and Outlook

In this chapter the results and its implications that are obtained through the corresponding Implementation and Experimental study done in Zephyr RTOS are discussed. Also further recommendations for future work, that is needed for the improvement of the deterministic Ethernet communication are discussed.

7.1 Discussions

The major challenges associated with the current distributed architecture of power stress testing setup was related to *scalability* and the limitations on deterministic communication faced by the *critical data*. The two research questions formalised in Chapter 1 clearly addresses these shortcomings of the current setup and proposed a detailed plan to overcome those in Chapter 3.

The increased *real-time* functionalities and open source community support provided by the Zephyr RTOS helped in choosing Zephyr as the target OS to be run on individual *control nodes*. The main challenge was to enable Ethernet communication between two Zephyr running *Infineon XMC4500* based *control nodes*. The complete implementation of XMC4500 based Zephyr Ethernet driver was done and tested against different type of communication protocols (e.g. TCP/IP, UDP/IP) for enabling communication infrastructure.

It was identified that a *topic*-based *Publisher-Subscriber* IoT messaging protocol like MQTT was best suited for solving the Scalability issue. The *Host PC* based centralised architecture was modified to accommodate an *MQTT broker* into the *Host PC* to enable M2M communication between *control nodes*. The proposed solution was verified through a comprehensive experimental study, where *control nodes* acted as *MQTT clients*, publishing and subscribing data through common *MQTT topics*. The

latency and jitter results collected for different QoS levels exposed by MQTT are also carefully analysed in Chapter 6. These results showed that an increased QoS for the TCP/IP based MQTT, increased the overall latency and jitter.

To address the deterministic *critical* data communication between *control nodes*, UDP/IP based MQTT-SN was proposed. The multicast feature offered by IPv4 layer along with the help of an intermediate Ethernet switch ensured direct *Publisher-to-Subscriber* communication between MQTT-SN *control* nodes. The jitter distribution based on the measurement data results showed that the *multicasted MQTT-SN* communication with the lowest QoS = -1 (*fire-and-forget*), contributed to the lowest possible jitter. The flexibility offered by *fire-and-forget* packets to bypass both MQTT-SN gate and MQTT broker during transmission was the main reason for the low jitter. This low jitter communication was selected as the benchmark for further deterministic experiments, and was also selected for carrying *critical data* between *control nodes*.

Zephyr RTOS used a single network transmission queue for transmitting Ethernet packets to the driver. The latency and jitter imposed by *non-critical* traffic on the *critical* traffic were next studied as mentioned in Chapter 5 for ensuring deterministic behavior. TCP/IP and UDP/IP based noise traffic was sent along with the *critical* traffic (MQTT-SN) to simulate the maximum congestion while transmitting. Chapter 6 showed without any traffic priority mapping, the *critical* data experienced huge jitter, compared to the benchmark case. Assigning a higher priority to *critical* traffic brought back the jitter to a minimum, which was almost the same as the benchmark. The interesting result was with TCP/IP based noise traffic. It was observed that the jitter measure for TCP/IP noise was higher compared to UDP/IP noise, even with priority mapping. This is because of the packet collision between *critical* packets and TCP ACK packets.

It was shown that modifying the current setup by introducing MQTT broker in *Host PC* enabled scalable communication for *control nodes* and its ease of use was verified during the experimental study. Introducing a socket level priority mapping (IEEE 802.1Q implementation of Zephyr as described in Section 2.4.3.5) for the selected low-jitter MQTT-SN *critical* traffic enabled us to ensure deterministic guarantees (that is by ensuring minimum jitter is maintained even when the network transmit queues are congested with non-critical data). This work clearly gives meaningful answers to the research questions identified and serve as an initial work toward more advanced deterministic Ethernet communication.

7.2 Future work

The results obtained when priority-mapped *critical* traffic is mixed with TCP/IP based *non-critical* traffic (Section 6.2.2) indicated that around 16 % of the *critical* packets experienced a higher jitter. It was observed that the collision from pending TCP-based handshaking ACK packets caused the higher jitter. These collisions could be avoided if both *critical* and *non-critical* traffics are transmitted in separate TDMA like time-slotted schedule (e.g. IEEE 802.1 Qbv). The *worst-case* time that is needed for completing the TCP *handshakes* should be accounted for when modeling the schedule. The *critical* traffic could be transmitted collision free, when it's scheduled after this *worst-case* upper bound reserved as a *guard-band*.

The jitter experienced by the *critical* traffic (MQTT-SN) when it's congested with TCP has been implemented in this work. It is recommended to further extend the experimental study such that *non-critical* MQTT data (TCP based) is mixed with the MQTT-SN *critical* data traffic. Since the handshake mechanisms of the MQTT with different QoS levels is higher when compared to TCP, implementations of advanced traffic shaping and stream reservations will be required for these traffics to co-exist in a deterministic manner.

The current implementation for the deterministic traffic doesn't account for the starvation experienced by low-priority *non-critical* traffic. The *non-critical* traffic could be prevented from transmitting indefinitely if the high-priority *critical* traffic keeps filling the transmit network queues. A sudden *burst* of traffic which could potentially fill up the buffer in the switch along the path of transmission is also possible with the current design. Thus smoothing out the traffic having different priorities or traffic-classes across the time-domain are needed to overcome the buffer overflow conditions. The standards defined by IEEE 802.1 Qav could be adapted for implementing a *credit-based shaper*, which would smooth out the traffic and prevent low-priority *non-critical* traffic from starving.

To achieve a minimum latency, Ethernet MAC IP with dedicated hardware queues is recommended. Advanced microcontrollers like Infineon AURIX™ having multiple hardware transmit/receive queues for the Ethernet MAC could be a possible candidate. Also the future implementations of IEEE 802.1 Qbv (*time-aware shaper*) and IEEE 802.1 Qav (*credit-based shaper*) to achieve improved deterministic traffic is best suited with dedicated hardware queues for optimal performance.

The time synchronisation accuracy of the implemented PTP driver could be improved to sub-microsecond to achieve desired performance. This is critical because

all the *control nodes* and the ethernet switch need to be synchronised to the same time-domain. The time synchronised nodes can then share the same accurate traffic time schedules (generated from Qbv and Qav shapers) between them and behave deterministically. The sub-optimal performance of the current PTP driver could be improved by making use of a real-time capable switch with advanced PTP capabilities. The switch will allow capturing PTP network packets for debugging, correcting the PTP path delay by attaching switch *residence time* etc.

Bibliography

[1] B. Steinwender, S. Einspieler, M. Glavanovics, and W. Elmenreich, "Distributed power semiconductor stress test & measurement architecture," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, 2013, pp. 129–134. DOI: 10.1109/INDIN.2013.6622870.

[2] "Ethernet," Wikipedia. (Jun. 2022), [Online]. Available: https://en.wikipedia.org/wiki/Ethernet (visited on 2022-07-05).

[3] "MQTT: The Standard for Iot Messaging," Eclipse. (2022), [Online]. Available: https://mqtt.org/ (visited on 2022-07-03).

[4] "Eclipse Mosquitto - An open source MQTT broker," Eclipse. (2022), [Online]. Available: https://mosquitto.org/ (visited on 2022-07-05).

[5] "MQTT For Sensor Networks (MQTT-SN) Protocol Specification, Version 1.2," International Business Machines Corporation (IBM). (Nov. 2013), [Online]. Available: https://www.oasis-open.org/committees/download.php/66091/MQTT-SN_spec_v1.2.pdf (visited on 2022-07-01).

[6] "Eclipse Paho MQTT-SN C/C++ client for Embedded platforms," Eclipse. (Dec. 2021), [Online]. Available: https://github.com/eclipse/paho.mqtt-sn.embedded-c (visited on 2022-06-14).

[7] "Unified Architecture," OPC Technologies. (2022), [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/ (visited on 2022-06-29).

[8] "CoAP, RFC 7252 Constrained Application Protocol." (2016), [Online]. Available: https://coap.technology/ (visited on 2022-06-29).

[9] "OMA LightweightM2M Overview," Open Mobile Alliance. (2022), [Online]. Available: https://technical.openmobilealliance.org/Overviews/lightweightm2m_overview.html (visited on 2022-07-13).

[10] "Avionics Full-Duplex Switched Ethernet," Wikipedia. (Mar. 2022), [Online]. Available: https://en.wikipedia.org/wiki/Avionics_Full-Duplex_Switched_Ethernet (visited on 2022-07-03).

[11] "TTEthernet," Wikipedia. (May 2022), [Online]. Available: https://en.wikipedia.org/wiki/TTEthernet (visited on 2022-07-03).

[12] K. B. Stanton, "Distributing Deterministic, Accurate Time for Tightly Coordinated Network and Software Applications: IEEE 802.1AS, the TSN profile of PTP," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 34–40, 2018. DOI: 10.1109/MCOMSTD.2018.1700086.

[13] "802.1AS - Timing and Synchronization," IEEE. (Nov. 2010), [Online]. Available: http://www.ieee802.org/1/pages/802.1as.html (visited on 2022-07-07).

[14] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," IEEE. (2008), [Online]. Available: https://standards.ieee.org/ieee/1588/4355/ (visited on 2022-07-13).

[15] "802.1Qat - Stream Reservation Protocol," IEEE. (Jun. 2010), [Online]. Available: http://www.ieee802.org/1/pages/802.1at.html (visited on 2022-07-07).

[16] "802.1Qav - Forwarding and Queuing Enhancements for Time-Sensitive Streams," IEEE. (Dec. 2009), [Online]. Available: http://www.ieee802.org/1/pages/802.1av.html (visited on 2022-07-07).

[17] J. L. Messenger, "Time-sensitive networking: An introduction," *IEEE Communications Standards Magazine*, vol. 2, no. 2, pp. 29–33, 2018. DOI: 10.1109/MCOMSTD.2018.1700047.

[18] "802.1Qbv - Enhancements for Scheduled Traffic," IEEE. (Nov. 2016), [Online]. Available: http://www.ieee802.org/1/pages/802.1bv.html (visited on 2022-07-08).

[19] "Time-Sensitive Networking," Wikipedia. (Jun. 2022), [Online]. Available: https://en.wikipedia.org/wiki/Time-Sensitive_Networking#IEEE_802.1Qat_Stream_Reservation_Protocol (visited on 2022-07-08).

[20] "802.1Q-2014 - Bridges and Bridged Networks," IEEE. (2014), [Online]. Available: https://standards.ieee.org/findstds/standard/802.1Q-2014.html (visited on 2022-07-09).

[21] "Zephyr®Project," Zephyr. (2022), [Online]. Available: https://www.zephyrproject.org/ (visited on 2022-05-13).

[22] "Device Driver Model," Zephyr. (2022), [Online]. Available: https://docs.zephyrproject.org/latest/kernel/drivers/index.html (visited on 2022-07-13).

[23] "XMC4500, Microcontroller Series for Industrial Applications, Reference Manual V1.6 2016-07," Infineon. (Jul. 2016), [Online]. Available: https://www.infineon.com/dgdl/Infineon-xmc4500_rm_v1.6_2016-UM-v01_06-EN.pdf?fileId=db3a30433580b3710135a5f8b7bc6d13 (visited on 2022-07-10).

[24] "OXYGEN - dewetron's intuitive measurement software," Dewetron. (2022), [Online]. Available: https://www.dewetron.com/products/oxygen-measurement-software/ (visited on 2022-06-01).

[25] "Antmicro's work with Time Sensitive Networking support in the Zephyr RTOS," Antmicro. (Sep. 2019), [Online]. Available: https://antmicro.com/blog/2019/09/time-sensitive-networking-in-zephyr/ (visited on 2022-06-11).

[26] "LinuxPTP Project," Sourceforge. (2022), [Online]. Available: http://linuxptp.sourceforge.net/ (visited on 2022-06-21).

Acronyms

ACK	Acknowledge
API	Application Programming Interface
ARP	Address Resolution Protocol
AVB	Audio Video Bridging
BMCA	Best Master Clock Algorithm
BSD	Berkeley Software Distribution
CAN	Controller Area Network
CBS	Credit Based Shaper
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSV	Comma-Separated Values
DMA	Direct Memory Access
DUT	Device Under Test
FCS	Frame Check Sequence
FIFO	First In First Out
GPIO	General Purpose Input Output
gPTP	generic Precision Time Protocol
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IIoT	Industrial Internet of Things
IoT	Internet of Things
IP	Intellectual Property
IP	Internet Protocol
ISR	Interrupt Service Routine
KAI	Kompetenzzentrum Automobil- und Industrie-Elektronik

L2	Layer 2
LAN	Local Area Network
LWM2M	Light Weight M2M
M2M	Machine-to-Machine
MAC	Media Access Control
MAN	Metropolitan Area Network
MDC	Management Data Clock
MDIO	Management Data Input/Output
MII	Media Independent Interface
MQTT	Message Queuing Telemetry Transport
MQTT-SN	MQTT For Sensor Networks
NIC	Network Interface Card
NVIC	Nested Vector Interrupt Controller
OASIS	The Organization for the Advancement of Structured Information Standards
OAuth	Open Authorisation
OMA	Open Mobile Alliance
OPC UA	Open Platform Communications United Architecture
OS	Operating System
OSI	Open Systems Interconnect
PC	Personal Computer
PHY	Physical Layer
PPS	Pulse Per Second
PTP	Precision Time Protocol
QoS	Quality of Service
RAM	Random-Access Memory
RMII	Reduced Media Independent Interface
ROM	Read-Only Memory
RTOS	Real-Time Operating System
RTT	Real Time Transfer

SFD	Start of Frame Delimiter
SMA	Station Management Agent
SMI	Station Management Interface
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SYNC	Synchronization
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TLS	Transport Layer Security
TSN	Time-Sensitive Networking
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
UniTn	University of Trento
URL	Uniform Resource Locator
USB	Universal Serial Bus
VLAN	Virtual-LAN
WAN	Wide Area Network