FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	A Study on Time-Sensitive Networking Integration within PikeOS RTOS
Student:	Bc. Jakub Šatoplet
Supervisor:	doc. RNDr. Ing. Petr Zemánek, CSc.
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2024/2025

Instructions

The objective of this master's thesis is to develop a device driver for the NXP i.MX8M Plus ENET_QOS Ethernet controller within the PikeOS Driver Development Kit (DDK) framework. This project requires a comprehensive understanding of hardware, software, and real-time networking concepts, as it involves designing and implementing a critical component for real-time embedded systems.

The theoretical part of the thesis must introduce basic concepts of real-time operating systems (RTOS) and focus on time-sensitive networking (TSN). This focus includes introducing real-time networking scenarios and analysing strategies and protocols used in TSN.

Analyse the Embedded Starterkit STKa8MPxL Evaluation kit and the NXP i.MX 8M Plus processor. Examine the development of a driver for the ENET_QOS Ethernet controller within the PikeOS DDK framework and functionalities of its Network class. Implement support for these functionalities for the ENET_QOS controller. Identify the controller's extended functionalities, such as VLAN and TSN, and analyse how they could be integrated into PikeOS.

The expected outcome is a functional device driver for the ENET_QOS Ethernet controller. This driver must implement basic functionalities such as packet send and receive, PHY link control, frame filtering, and multicast handling. Additionally, an interface for the

---

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 16 November 2023 in Prague.

controller's TSN features should be designed, with a discussion of how it could be integrated into the PikeOS DDK Ethernet class.

Leverage these references:
- NXP i.MX 8M Plus Applications Processor Reference Manual,
- PikeOS Device Driver Programming Reference Manual,
- Linux Driver for EQOS,
- SYSGO's Architecture and Study of TSN on ls1028a target,
- other resources related to the controller and PikeOS.

Master's thesis

A STUDY ON TIME-SENSITIVE NETWORKING INTEGRATION WITHIN PIKEOS RTOS

Bc. Jakub Šatoplet

Faculty of Information Technology
Department of Theoretical Computer Science
Supervisor: doc. RNDr. Ing. Petr Zemánek, CSc.
May 5, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Šatoplet Jakub. *A Study on Time-Sensitive Networking Integration within PikeOS RTOS*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

I am incredibly grateful to my master's thesis supervisor, doc. RNDr. Ing. Petr Zemánek, CSc. and the PikeOS team for allowing me to work on such a fascinating topic. Your guidance and support have been invaluable. Working with the PikeOS team has been inspiring, and I'm thankful for the opportunity to contribute to cutting-edge developments in the field.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 9/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work, but only for non-profit purposes. This authorisation is unlimited in time, territory and quantity.

In Prague on May 5, 2024 .....

Abstrakt

Cílem diplomové práce je studie integrace časově citlivých sítí do operačního systému reálného času PikeOS. V analytické části práce jsou představeny operační systémy reálného času a systém PikeOS. Dále představuje základní koncepty časově citlivých sítí s důrazem na synchronizaci času pomocí protokolu Precision Time Protocol. Implementační kapitoly vysvětlují, jak jsou strukturovány ovladače systému PikeOS, se zaměřením na Ethernetové ovladače. Jejich hlavním výstupem je ovladač pro Ethernetový řadič ENET_QOS. Kromě základních funkcí přenosu rámců patří mezi vlastnosti ovladače pokročilé filtrování rámců se značkami IEEE 802.1Q, které představilo návrh nového standardního rozhraní, zpracování vícesměrového vysílání a integrace protokolu Precision Time Protocol. Testování ovladače ověřilo jeho správnost a naměřilo chybu integrované časové synchronizace v očekávaném rozsahu desítek mikrosekund.

Klíčová slova TSN, PTP, IEEE 1588, PikeOS, RTOS, Ethernet, počítačové sítě

Abstract

The objective of the master's thesis is a study on the integration of time-sensitive networking into the PikeOS real-time operating system. The analytical part of the thesis introduces real-time operating systems and the PikeOS. It then presents core concepts of time-sensitive networks, emphasizing time synchronization using the Precision Time Protocol. The implementation chapters explain how PikeOS drivers are structured, focusing on Ethernet drivers. Their core outcome is a driver for the ENET_QOS Ethernet controller. In addition to the essential frame transfer functionalities, the driver's features include advanced filtering of IEEE 802.1Q tagged frames, which presented a new standard API proposal, multicast handling and the Precision Time Protocol integration. Testing the driver verified its correctness and measured the error of the integrated time synchronization in the expected range of tens of microseconds.

Keywords TSN, PTP, IEEE 1588, PikeOS, RTOS, Ethernet, computer networks

Contents

List of Figures

List of Tables

List of Code Listings

List of Abbreviations

A/V	audio/video
ABS	anti-lock braking system
ACL	access conrol list
ADAS	advanced driver-assistance system
AHB	AMBA High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
API	application programming interface
ARB	arbitrary
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
ASP	architecture support package
AVB	audio/video bridging
AXI	Advanced eXtensible Interface
BC	boundary clock
BIPM	Bureau International des Poids et Mesures
BMCA	Best Master Clock Algorithm
BSP	board support package
C-VLAN	customer virtual local area network
CAN	controller area network
CDC	clock domain crossing
CPU	central processing unit
CRC	cyclic redundancy check
CSMA	Carrier-sense multiple access
CSR	control and status register
cTE	constant time error
DDK	Driver Development Kit
DDR	double data rate
DMA	direct memory access

DPLL	digital phase locked loop
DSC	Data Set Comparison Algorithm
dTE	dynamic time error
E2E	end-to-end
EEC	Ethernet equipment clock
ESMC	Ethernet Synchronization Messsaging Channel
EST	Enhancements for Scheduled Traffic
ExtFP	External File Provider
FCS	frame check sequence
FFD	fractional frequency deviation
FIFO	first-in-first-out
FIR	finite impulse response
FS	file system
GM	grandmaster
GNSS	global navigation satellite system
GPIO	general purpose input output
gPTP	generalized Precision Time Protocol
I/O	input/output
I2C	Inter-Integrated Circuit
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IDE	intergrated development environment
IEEE	Institute of Electrical and Electronics Engineers
IERS	International Earth Rotation and Reference Systems
IIR	infinite impulse response
IOMMU	input/output memory management unit
IoT	Internet of Things
IPC	inter-process communication
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
ITU	International Telecommunication Union
KiB	kibibyte
LACP	Link Aggregation Control Protocol
LAN	local area network
LC	local clock
LKM	loadable kernel module
LPC	local PTP clock
LPDDR	low power double data rate
LPI	low power interface
MAC	media access control

MAN metropolitan area network
max|TE| maximum absolute time error
MCP maximum controlled priority
MDC management data clock
MDIO management data input/output
MII media independent interface
MMU memory management unit
MSB most significant bit
MTIE maximum time interval error
MTL media access control transaction layer
MTS maximum transfer size
MTU maximum transfer unit
NET DDK Network Class
NIC network interface controller
NTP Network Time Protocol
NUI network unique identifier
OC ordinary clock
OCXO oven-controlled crystal oscillator
OS operating system
OUI organizationally unique identifier
P2P peer-to-peer
PDU protocol data unit
PDV packet delay variation
PHY physical layer
PI proportional-integral
PLL phase locked loop
PLS physical signaling sublayer
ppb parts per billion
ppm parts per million
PPS pulse-per-second
PRE preamble
PropFS Property File System
PRTC primary reference time clock
PSP platform support package
PSSW PikeOS System Software
PTP Precision Time Protocol
QoS quality of service
RGMII reduced gigabit media independent interface
RMII reduced media independent interface
RMS root mean square

ROM	read only memory
RTC	DDK Real-time Clock Class
RTOS	real-time operating system
RX	receive
S-VLAN	service virtual local area network
SBC	single-board computer
SBUF	shared buffer communication
SD	State Decision Algorithm
SDH	Synchronous Digital Hierarchy
SDR	single data rate
SI	International System of Units
SOF	start-of-frame
SSM	Synchronization Status Message
SyncE	Synchronous Ethernet
TAI	Temps Atomique International
TC	transparent clock
TCP	Transmission Control Protocol
TCXO	temperature-compensated crystal oscillator
TDEV	time deviation
TDM	time-division multiplex
TE	time error
TLV	type-length-value
ToD	time-of-day
TSN	time-sensitive networking
TTL	time-to-live
TX	transmit
UART	universal asynchronous receiver-transmitter
UDP	User Datagram Protocol
UNIX	UNiplexed Information Computing System
UTC	Coordinated Universal Time
UX	user experience
VCO	voltage-controlled oscillator
VCXO	voltage-controlled crystal oscillator
VLAN	virtual local area network
WAN	wide area network
WR	White Rabbit
XNU	X is Not Unix
XO	crystal oscillator

Chapter 1

Introduction

Real-time embedded systems have become integral to many aspects of modern life. Such systems reside in everything we encounter daily, from modern jet airliners and automobiles to household appliances and mobile devices. Industries such as factory automation, telecommunications, and vehicle navigation rely heavily on such systems. Recent advancements in information and communication technology and the Internet of Things (IoT) emergence have driven the real-time embedded application market to new heights.

These technologies' ubiquitous presence and application have brought an extreme number of such systems online. In cases like factory automation, fleets of hundreds or even thousands of independent robots must cooperate. Modern wireless telecommunication technologies such as 5G enable this. However, to build modern telecommunication networks and even reach such a level of coordination, synchronized time is of the essence. If one robot drops an item, expecting the other to catch it and transport it elsewhere, care must be taken to ensure both have an identical sense of when this should happen. Nevertheless, synchronization has been a fundamental necessity since the emergence of digital networks in the 1970s to transfer voice. All these examples are what time-sensitive networking conveys and tries to solve.

This work delves into what synchronization and even time itself mean. Diagnosing synchronization issues is challenging as they often manifest as logic design problems rather than clocking errors. Thus, robust network design is essential to mitigate sporadic outages and data loss. These topics are explored in the context of a real-time operating system called PikeOS.

Chapter 2 introduces real-time operating systems and PikeOS. The third chapter delves into time synchronization in computer networks and the Precision Time Protocol. Chapters 4 and 5 describe the integration of a new Ethernet controller into PikeOS, including time synchronization. The last chapter of content six is dedicated to testing the driver's functionalities.

Chapter 2

Real-time systems

This chapter defines the fundamental terms of real-time systems. The PikeOS real-time operating system is situated within the context of these terms, and its core concepts are introduced. This facilitates understanding its significance within the domain of real-time operating systems.

2.1 Embedded systems

A microcomputer *embedded* system is a system which is incorporated into a more extensive system. It is designed for a specific task. That is in contrast to a microcomputer general purpose system, which typically has multiple functions. An example of such a system is the anti-lock braking system (ABS) in automobiles. It is a safety system which tries to retain directional control of a vehicle when its wheels lock due to a sudden application of brake pedal pressure. That is its sole purpose. Such systems are reactive systems by nature. [1] They can be divided by scale into three categories:

Small-scale embedded systems typically function without an operating system. They are most commonly implemented in assembly on eight or 16-bit microcontrollers. An example of such a system is a television remote control;

Medium-scale systems typically include an operating system. They are usually implemented using more traditional tools like the C language on more complex microprocessors. A vending machine is an example of a medium-scale system;

Large-scale systems are extremely complex applications of cutting-edge hardware and software co-design. Notable examples are the landing gear assembly of a modern jet airliner or advanced driver-assistance systems (ADAS) in vehicles. [1]

Embedded systems are additionally differentiated into two groups:

Non-real-time system performs based on internal triggers or external stimuli. It must satisfy a given level of quality of service (QoS). An example is the television remote;

Real-time system must deliver correct results in a given time. This is called the task's *deadline*. ABS and ADAS are both an example. [1]

Real-time systems are further divided into two groups:

Soft real-time system may still deliver its result after the deadline, which results in degraded performance. Multimedia applications are an example;

Hard real-time system is one whose result is useless if delivered after its deadline. This may have catastrophic consequences depending on the system. An airbag deployment system is an example. [1]

Note that non-real-time systems may also have timing constraints on result delivery. They still try to provide a quality user experience (UX). [1]

2.2 Operating systems

On the highest level of abstraction, a computer system consists of hardware, software and data. The operating system is a control program which acts as an allocator of these resources. It decides their efficient use while facing conflicting requests. There is not one accepted definition of what an operating system consists of. Some operating systems include vendor-provided applications taking up gigabytes of space; others do not even include full-screen access in kilobytes. A commonly followed definition is that an operating system is the only always running program called the *kernel*. [2]

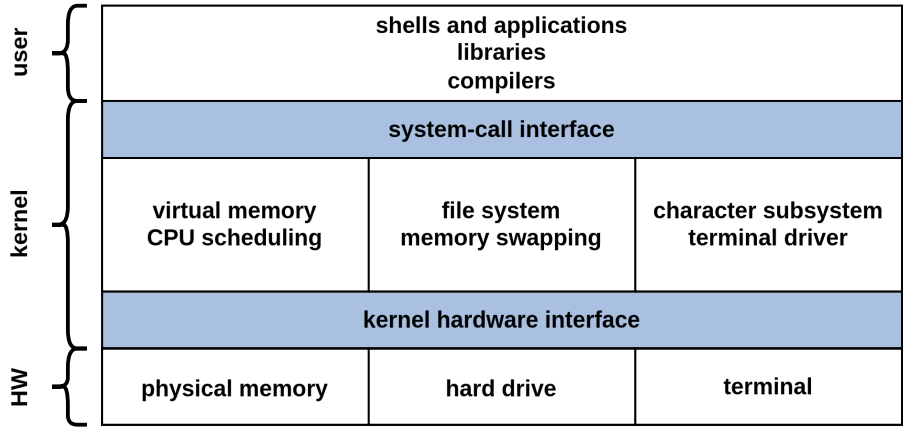
Modern kernels are complex systems. They handle many duties, from scheduling tasks on the central processing unit's (CPU) cores, managing memory, and providing access to input/output (I/O) devices to networking and even virtualizing other operating systems. This requires careful design and engineering to keep such systems easily modifiable, extensible and properly functioning. [3]

2.2.1 Monolithic kernel

A monolithic structure is the simplest way of organizing a kernel. The kernel runs in a single address space, providing all its functionality from a single static binary. [2]

The most notable example of a monolithic kernel is the UNiplexed Information Computing System (UNIX). It consists of two parts: the system programs and the kernel. The kernel provides process scheduling, memory management, file system, and other

functions through system calls. Functionality is extended using device drivers. This provides a layered design where the kernel lies between the system call interface and the hardware. This structure is depicted in Figure 2.1. The Linux kernel is structured highly similarly. It extends this with a modular design through loadable kernel modules (LKM), which can expand the kernel's functionality at runtime. [2]



■ **Figure 2.1** UNIX kernel structure. The system programs are at the top. They are connected to the kernel through the system call interface. The kernel below this interface communicates with the hardware through the kernel interface. [2]

The main disadvantage of monolithic kernels is the limited separation of their components. Internal and third-party modules share the same memory space. This may lead to fault propagation and even system-wide failures and memory corruption. Testing components in isolation is also inherently difficult. All of these reasons lead to increased verification complexity of modules because interaction with other parts of the kernel must be taken into account. [4]

On the other hand, communication inside the kernel itself is remarkably swift. Also, the system-call interface used by system programs has small overhead. Therefore, there is apparent efficiency and performance benefit, which explains its wide use in projects such as the Linux kernel. [2]

2.2.2 Microkernel

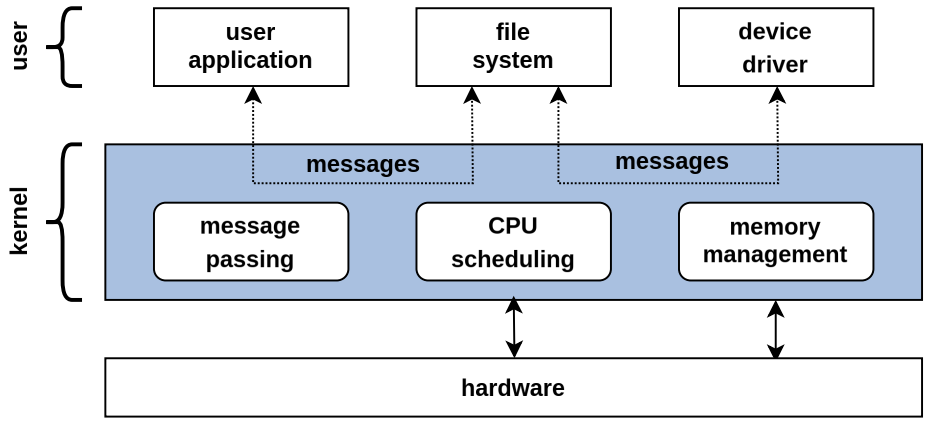
A *microkernel* is structured as small as possible. All non-essential components are removed from the kernel and implemented as user space applications. This includes components such as device drivers and file systems. Each of these applications has its own address space. This results in a small binary with a clear separation of all components. First example of this architecture was the Mach kernel. [2]

The kernel implements only the critical functionalities for running the user space programs. These include CPU scheduling and memory management. The programs

communicate with the kernel and between themselves through a *message-passing interface*. This architecture is illustrated in Figure 2.2. [2]

The kernel provides the interface and acts as a router for these messages. This leads to a simple solution to access control because all messages pass through the kernel. However, this leads to a significant performance penalty. The messages need to be transferred from one address space to the address space of the destination process. The kernel may also need to switch contexts between these processes to deliver a message. [2]

A clear advantage of this structure is fault tolerance. Because the programs reside in their own address spaces, any failure is limited to the program, excluding inter-process dependencies. Memory corruption between modules also cannot happen. The kernel itself is simpler than a monolithic kernel. This leads to a smaller chance of introducing bugs during its modifications because they tend to be small. Porting the kernel to other architectures is also simplified. All of these properties lead to easier verification of components because they can be tested in isolation. [4]



■ **Figure 2.2** Microkernel architecture. Most components, such as device drivers and file systems, are separated into user-space partitions. The kernel contains only the most critical components and a message-passing interface. [2]

2.2.3 Hybrid kernel

Monolithic kernels provide performance benefit while being less fault tolerant. Microkernels provide high fault tolerance and modularity while taking a performance hit. In practice, this leads to no operating system strictly following one of these architectures. They combine features and decisions made in both of these architectures composing a hybrid system. The Linux kernel is monolithic but modular through the use of LKMs. [2]

Nevertheless, the Darwin operating system is the most notable example of a hybrid system. Darwin is the base of Apple's macOS, iOS, and other operating systems for its products [2]. The core of Darwin is the X is Not Unix (XNU) kernel, composed of the Carnegie Mellon University Mach kernel, components from the FreeBSD operating sys-

tem and IOKit. The IOKit is an object-based C++ application programming interface (API) used for writing device drivers. [5]

2.2.4 Real-time kernel

Real-time kernels are the core of real-time operating systems (RTOS). They differ from general-purpose kernels in the necessity for specific system response times. Two requirements must be fulfilled by their design. Firstly, the kernel must have predictable timing behaviour of all its services. This means that an upper bound on the execution time of the operation must be known. These services include system call and interrupt handling. The second requirement is that the kernel must manage this timing. The task scheduler must consider the task deadlines when deciding what to run next. [1]

2.3 PikeOS

The PikeOS combines an Eclipse-based integrated development environment (IDE), a virtualization platform and a real-time operating system. It focuses on applications requiring certification that target harm prevention, safety-critical, or demand protection from malicious activities, security-critical. Examples of industries employing such applications are medical, aerospace and automotive industries. It uses so-called software partitions to spatially and temporally segregate different applications running on the system. This feature allows PikeOS to concurrently execute multiple applications on one system, where each may require a different certification level. [6]

The PikeOS Core implements the means for this separation, also called the PikeOS Hypervisor. The applications inside the software partitions can even be complete guest operating systems such as Linux. [6] The selection of implemented features includes:

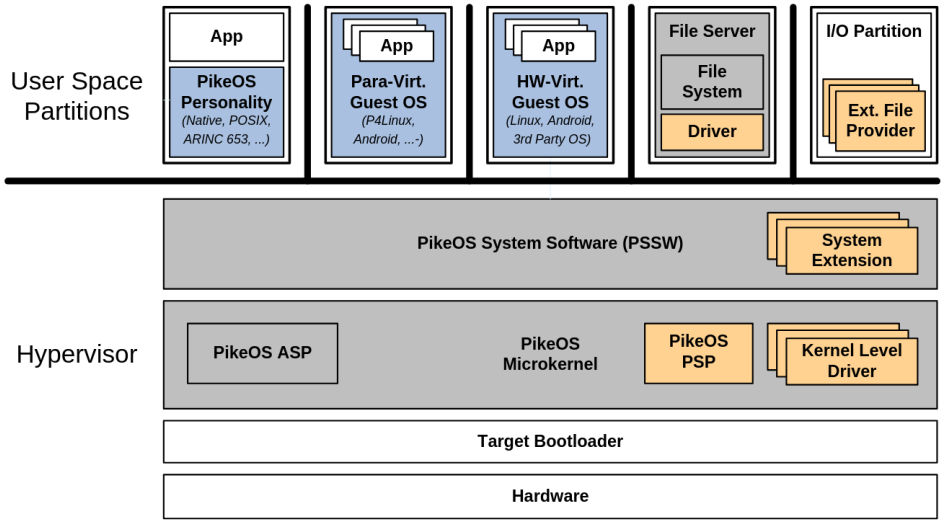
- Critical applications may run in their process, including their virtual address space. Hardware memory protection using a memory management unit (MMU) is possible. This includes protection of data memory corruption by direct memory access (DMA) peripherals using input/output MMU (IOMMU);
- Scheduling methods used are deterministic. Strict priority-based scheduling is used with a possible prevention of priority inversion. This includes the use of so-called time partitioning, which enables the allocation of processor time during build time;
- Interrupt handling provides a means to ensure the correct and safe execution of critical applications even during interrupt reception and handling. [6]

2.3.1 Architecture

PikeOS is designed modularly to support new architectures easily and adapt to the needs of individual projects. Its architecture is illustrated in Figure 2.3. [6] The PikeOS Core consists of two components:

PikeOS Microkernel is a real-time microkernel. It consists of a generic part, a processor architecture support package (ASP) and a platform support package (PSP). It is the only component in the system which runs with supervisor privileges. The main selection of features provided by it is hardware abstraction, address space separation, time partitioning, task scheduling, interrupt handling, timer handling, and message passing primitives;

PikeOS System Software (PSSW) is the first user space application started by the kernel. Its purpose is access control enforcement and system configuration. It initializes partitioning and inter-partition communication. After this, it stays active as a server that provides health monitoring services, file system services, process management and partition communication. [6]



■ **Figure 2.3** PikeOS architecture. The application partitioning is visible at the top. The PikeOS Core layering is depicted bellow with both the PSSW and the microkernel at the bottom. Target bootloader is also shown which is out of scope of PikeOS. [6]

2.3.2 Resource partitions

Resource partitions, also called virtual machines, are the basic security tool used for application separation in PikeOS. Each partition defines system resources to which the

applications inside the partition have access. Applications running in one partition cannot probe other partitions and are thus unaware of their applications. [6]

A configured number of partitions is created during boot. At startup, all partitions are empty. The only exception is the *global data* partition. The kernel initializes it to run the PSSW. Partitions cannot be deleted, but they may get stopped or restarted with a new set of applications. However, this process does not return the partition resources to the system. Returning resources to the system is impossible after their assignment. [6] Resources which need to be statically configured for use in a partition are:

Processes which should be started in this partition by the PSSW;

Communication ports for a message passing based inter-partition communication;

Files to which the applications should have access through the PSSW. Each file must be listed in the file access conrol list (ACL). Pathnames in PikeOS are of the form: `prefix:filename`. The prefix selects the file provider of the given filename. Wildcard access may be granted for simplification;

Memory and I/O resources granted. Memory is available to the application right after startup. If the partition is restarted, the memory is not returned to the system, and the application has access to the same memory as before. This ensures predictive application behaviour and prevents malicious data exchange through physical page reassignment. The memory size for application use must be configured. This includes the memory needed to load the application binary into and fulfil any dynamic allocation needs with. [6]

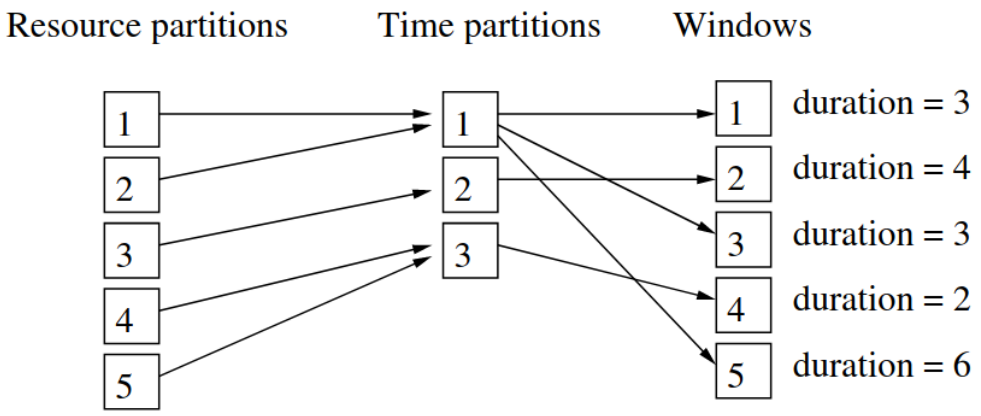
2.3.3 Time partitions

CPU time allocation is done using *time partitioning*. This can be used to prevent faulting threads from stealing all processor time, prevent thread starvation and ensure that partitions get hold of the CPU for a defined amount of time. [6]

In the simplest form, each resource partition gets assigned to a specific time partition. This is, however, not mandatory. Resource and time partitions are independent of each other. Multiple partitions may belong to the same time partition. Under certain circumstances, even threads of one partition may belong to different time partitions. [6]

An important fact is that the time window, the execution duration, is not a property of the time partition. Each time partition is assigned to one or more time windows. The time partition is then *activated* during scheduling. This means that the scheduler considers the time partition threads. [6]

The resource and time partition mapping can be done per-core on multicore systems. Example of the partition mapping is illustrated in Figure 2.4. [6]



■ **Figure 2.4** PikeOS time partition configuration. Illustrated is the mapping of five resource partitions into three time partitions. Each time partition is then mapped into at least one of the five time windows with a specified duration. [6]

2.3.3.1 Time partition zero

In PikeOS, there can be up to two time partitions active. The first may be any configured time partition, and the second is always the *time partition 0*. This time partition has the property of always being considered for scheduling. If two partitions are active, the scheduler looks at the thread with the highest priority from time partition 0 and for the thread with the highest priority from the other partition. It then chooses the higher priority thread. In case of a priority tie, it prefers the time partition 0. If no other time partition is active, only time partition 0 is considered. This design allows PikeOS to handle two scheduling situations effectively. [6]

Firstly, time partition 0 may handle background tasks when other application threads in any active time partition are idle. This helps to prevent wasting computational resources, predominantly the CPU. [6]

The important consequence of this design decision is, however, safety-critical applications. These applications include health monitoring and response to external events. The partition handling such applications may be assigned to time partition 0 with the highest maximum controlled priority (MCP) value in the whole system. This ensures that it can preempt any other thread when required. [6]

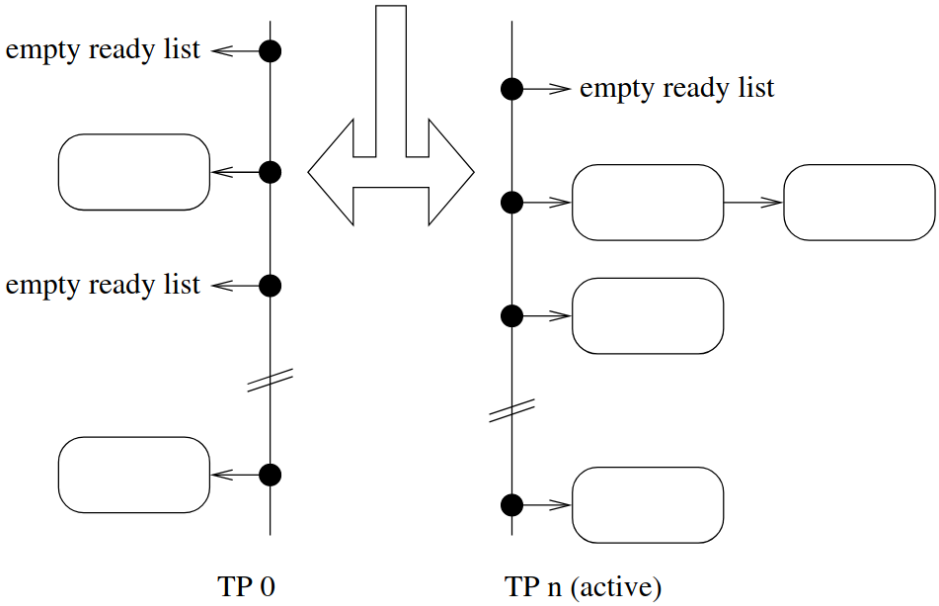
2.3.4 Scheduling

Processes in PikeOS are user applications started inside a partition by the PSSW partition daemon. One or more processes may be started inside a partition. Each process has its properties, which include a name and an identifier, the command line it was started with, its virtual address space, the MCP value, the maximum number of child tasks and the maximum number of threads. A *task* is defined as an address space and threads

bound to it. A process is consequently a task. *Thread* in PikeOS is a schedulable entity with a defined priority. [6]

The algorithm employed for scheduling in PikeOS is a fixed priority scheduling algorithm with a first-in-first-out (FIFO) ordering of threads with the same priority. Threads are selected from the currently active time partitions. The currently running thread is called the *current* thread. [6]

Each priority level has its own thread ready list. When a thread becomes runnable, it is placed at the end of its priority list. If a thread changes priority, it is placed at the end of the new priority list. If it is current during this change, it is placed at the front of the new priority list. The ready lists are tracked per time partition and CPU core on multi-core systems. The scheduling ready lists are illustrated in Figure 2.5. This illustration also shows the constantly considered time partition 0 by the scheduler. [6]



■ **Figure 2.5** PikeOS scheduling with time partitioning. The ready lists for time partition 0 and one other active partition are illustrated. The highest priority threads, displayed as white boxes, are placed at the top. Because the time partition 0 thread has higher priority, the scheduler selects it to be executed next. [6]

Chapter 3

Time-sensitive networking

This chapter focuses on the analysis of time synchronization over computer networks and its application in time-sensitive networking. This study includes an overview of the IEEE 1588 Precision Time Protocol.

3.1 Time in computer networks

Ethernet has been the core of computer networking for over 50 years. Its prevalence is impacted by its steady evolution and improvement, reinforced by industry support. It is extended mainly by two working groups of the Institute of Electrical and Electronics Engineers (IEEE). The IEEE 802.3 handles the physical (PHY) and media access control (MAC) layers of Ethernet. The 802.1 is concerned with architectures of local area networks (LAN), metropolitan area networks (MAN) and protocols above the MAC layer. The most notable standard for this work is 802.1Q – *Bridges and Bridged Networks*. This standard defines MAC bridges, virtual LANs (VLAN) and Ethernet QoS support. [7]

Note that amendments to IEEE standards are released. These are marked as lowercase extensions of the standard's name. An example is the 802.1Qbv, which is an amendment of the 802.1Q. When standards are superseded by newer versions, these amendments are typically included in the new version. However, the original lowercase name is traditionally used for easier recognition when referring to the amendment. [8]

3.1.1 Best-effort networks

Before transferring audio/video (A/V) streams over computer networks, they have been optimized to provide *best-effort* delivery. This, in practice, meant providing the maximum throughput. For networks with light loads concerned with the average delay of data passing through, this solution truly works as the best one. On the other hand, it breaks down when *time* becomes the critical metric. [9]

This is the case for A/V streams. Human brains perceive audio coming before corresponding visuals as unnatural, but this is not the case the other way around. A/V transmissions require all audio channels to be at most 20 μs within each other. Video may come even 25 ms before its audio track, but the other way around the limit is 15 μs. Best-effort networks are not able to fulfil these requirements reliably. [9]

3.1.2 Time-sensitive networks

Two requirements must be satisfied in *time-sensitive* networks (TSN). Firstly, the wall clock of devices in the network must be the same on all nodes. This is important for phase locking, coordination and synchronization. Secondly, the data must arrive in a specific time window. Most commonly, a maximum delay is specified. [9]

The IEEE Audio/Video Bridging (AVB) task group of the 802.1 working group was tasked with developing mechanisms to solve these constraints. It was later renamed to the TSN task group. This was done because the TSN requirements far exceed the use case for AVB. [7] For example, the automotive industry shares these for its real-time ADAS systems reacting to many sensors. These are commonly connected using 10BASE-T1S Ethernet, replacing the controller area network (CAN) bus [10]. Academic research includes RTOSs with scheduling synchronized to global navigation satellite systems (GNSS) [11]. Furthermore, TSN is a critical attribute of modern cellular telecommunication networks [12].

3.2 Time-sensitive networking standards

The AVB group developed four core standards to meet the QoS goals:

802.1AS – *generalized Precision Time Protocol* (gPTP): PTP profile for any network technology based on the IEEE 802 architecture;

802.1Qav – *Forwarding and Queueing of Time-Sensitive Streams*: a traffic shaping mechanism using credit based shaping instead of static priorities;

802.1Qat – *Stream Reservation Protocol*: bandwidth reservation protocol;

802.1BA – *AVB Systems*: complete AVB system architecture. [9]

Today, more than 30 IEEE standards are defined for TSN networking, including amendments. These can be divided into four groups targeting synchronization, reliability, latency and resource management. [13] The TSN core standards, additionally to the AVB standards, include:

802.1Qbv – *Enhancements for Scheduled Traffic* (EST): a mechanism to drain frame queues inside network bridges synchronously based on a common gPTP time base;

802.1Qbu – *Frame Preemption*: a mechanism which allows frames to preempt other frames to improve latency and bandwidth utilization;

802.1Qca – *Path Control and Reservation*: a mechanism to set up multiple redundant paths between network nodes for improved reliability;

802.1CB – *Frame Replication and Elimination for Reliability*: a mechanism to eliminate redundant frame copies. [7]

3.3 Time synchronization

Even though some of the services provided by different categories of TSN standards may be used independently, synchronized time is the minimum required common property for any practical application of most of these standards. An example is the 802.1Qbv. [9] Three primary terms must be distinguished regarding computer networks synchronization: frequency, phase and time-of-day [12].

3.3.1 Frequency

The definition of *frequency* is the number of recurring event repetitions over a given unit of time. The standard unit used is the *Hertz* (Hz), which describes the event count over a second. The inverse of frequency is called *period* and defines the duration of one complete cycle. For example, a swinging pendulum with a frequency of 4 Hz has a period of 250 ms. Mathematical representation is: [12]

$$Frequency(f) := \frac{1}{Period(T)} \tag{3.1}$$

Any device which performs some action at a regular interval may be used as a frequency generator. Such a device is called a resonator. When this device is connected to an electrical circuit to continue this behaviour indefinitely, it is called an *oscillator*. [12]

Oscillators have their *nominal frequency* specified. That is the frequency at which the oscillator was designed to resonate. Because of external factors and manufacturing imperfections, the frequency generated will deviate from the nominal frequency. This deviation is called the *fractional frequency deviation* (FFD). The FFD units are parts per million (ppm) or parts per billion (ppb). [12] The FFD at time t is mathematically defined by the International Telecommunication Union (ITU) in its recommendation T-REC-G.810 – *Definitions and terminology for synchronization networks* as:

$$y(t) := \frac{f(t) - f_{nom}}{f_{nom}} \tag{3.2}$$

where $f(t)$ is the measured frequency and f_{nom} is the nominal frequency. [14]

This rate may also be looked at over time duration. For example, an FFD of +5 ppm means the signal accumulates 5 μs more every second, 50 μs over 10 seconds, against the reference frequency. This is called *frequency drift* with units of Hz/s. [12]

Frequency synchronization minimizes the presented frequency errors between two frequency sources. When a selected error threshold is reached, we say the two sources are frequency synchronized. This process is called *syntonization*. [12]

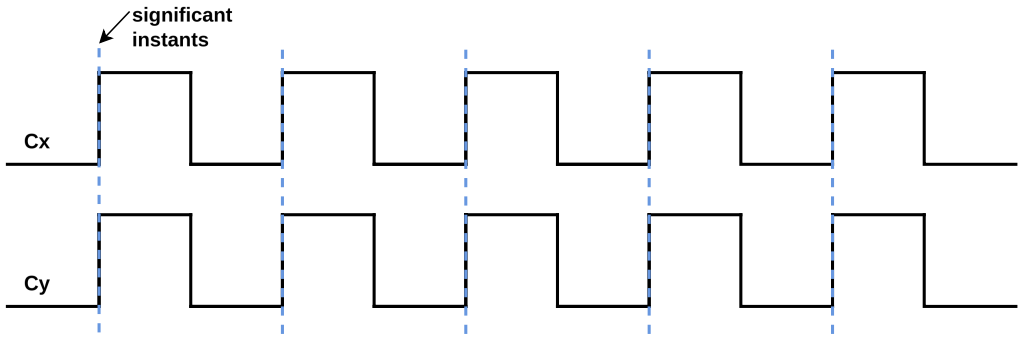
3.3.2 Phase

Phase is a term describing events happening simultaneously. It is not measurable by itself. Phase can be only measured as a relative offset to a known reference. For example, we can imagine two wall clocks, each having a second-hand ticking at a frequency of 1 Hz. The first hand started at time 0, and the second hand started 300 ms after the first one. The hands are syntonized, but they are not phase synchronized (phase aligned). The second hand has a *phase offset* of -300 ms against the first. Contrariwise, the first hand has a phase offset of +300 ms against the second. Mathematically, if $C_X(t)$ is the time reported by clock X, then the relative phase offset θ of clock X to clock Y is: [12]

$$\theta := C_X(t) - C_Y(t) \tag{3.3}$$

Note that syntonization is not required for phase synchronization. The point at which we sample the phase is chosen arbitrarily; if one hand would increase frequency to 4 Hz and the phase was sampled every fourth tick of this hand, phase synchronization may still be achieved. Nevertheless, syntonization and phase synchronization are tightly coupled because syntonization keeps the phases close. [12]

The sample point is called the *significant instant* and must be repeatable. Two sources are phase synchronized, or phase aligned, when their significant instants happen simultaneously. Phase alignment is illustrated in Figure 3.1. [12]

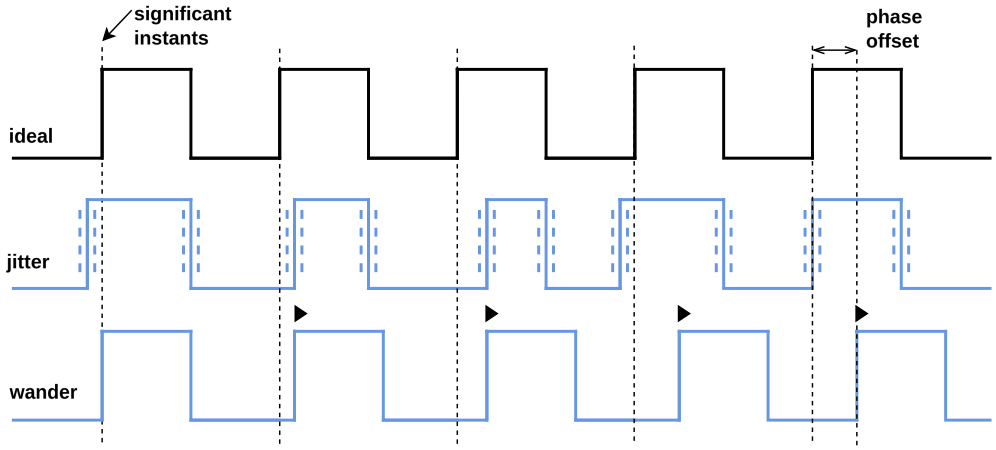


■ **Figure 3.1** Syntonized phase-aligned square wave clock signals. The blue lines indicate the significant instants for clock Cx and clock Cy. [12]

The phase is prone to two types of errors, *noise*, in the timing signal: long-term low-rate (less frequent) variations of the significant instant and short-term high-rate (more frequent) variations. These errors are illustrated in Figure 3.2. The clocking application must specify what low-rate and high-rate mean in its context. PTP specifies 0.1 Hz, whereas ITU-T-G.810 specifies 10 Hz as an example. [12] The two errors are:

Wander represents the long-term variations of the significant instants. It may be imagined as a long-term shifting of the significant instant in one direction. It cannot get filtered completely in practice;

Jitter represents the short-term variations of the significant instants. It may be imagined as a flickering of the edges of a square wave clock signal back and forth. Because it is short-term, it may get filtered out by sampling only the long-term trend of the sampled oscillator. [12]



■ **Figure 3.2** Clock signal jitter and wander. Depicted is an ideal signal at the top. In the middle is illustrated jitter as flickering back and forth against the reference. Wander drifting forward (slower than reference) is portrayed at the bottom. [12]

3.3.3 Time-of-day

The *time-of-day* (ToD) is the absolute time humankind agrees on. It can be defined as a value of some counter since a synchronization point called the *epoch*, which increments phase aligned and at the same frequency for everyone. The most common representation of ToD is textual. That is a string of characters describing some part of ToD, for example, the time and possibly the date. [12]

The value of time defined by the International System of Units (SI) is the *Second* (s). It has a precise definition at the Bureau International des Poids et Mesures (BIPM) of Paris, France, the International Bureau of Weights and Measures in English, website: [15]

> *The second, symbol s, is the SI unit of time. It is defined by taking the fixed numerical value of the caesium frequency Δv_{Cs}, the unperturbed ground-state hyperfine transition frequency of the caesium-133 atom, to be 9 192 631 770 when expressed in the unit Hz, which is equal to s^{-1}.*

There are two time scales commonly used by humankind:

Temps Atomique International (TAI) is the International Atomic Time in English. It is based on a weighted average of over 400 atomic clocks placed worldwide. The epoch defined is the 1 January 1958 00:00:00. It is a *monotonic* timescale. This means that it increases each second and never jumps forward or back. This property is why the general public does not use it. TAI is out of sync with the rotation of the Earth. That is out of sync with day and night;

Coordinated Universal Time (UTC) is a *discontinuous* timescale with the same epoch as TAI. The International Earth Rotation and Reference Systems (IERS) service inserts leap seconds every few years to align the UTC with the Earth's rotation within 0.9 seconds. This makes UTC aligned with day and night. Note that the difference between TAI and UTC is 37 seconds at the time of writing. [12]

3.4 Clocks

A *clock* is an oscillator with a counter, a timekeeping device. The oscillator supplies time interval, whereas the counter provides time indication. It has a nominal frequency, as specified by its oscillator, typically provided as a square wave signal on an output pin in electronics. Suppose its counter phase alignment is calibrated to some external time-of-day epoch second, such as the TAI epoch second. In that case, the pulse-per-second (PPS) signal may provide a square wave signal of configurable duration each epoch second. [12]

Three quality factors, directly affected by the underlying oscillator characteristics, denote the clock's performance. *Ageing* describes how the frequency generated changes over the clock's lifespan. *Stability* represents how repeatable the signal is, measured over time. Finally, *accuracy* depicts how much the clock time deviates from the calibrated reference. For example, a clock with high positive ppm FFD against some reference is not accurate, but if the FFD measures the same for an extended period, it is stable. The meaning of high and extended depends on the application the clock is used in. [12]

During operation, the clock generally transfers between five states [12]. This is illustrated in Figure 3.3. The states are:

Warm-up is the state each clock enters after a power loss. The oscillator needs time to stabilize based on external temperature, airflow, supplied voltage and other external factors;

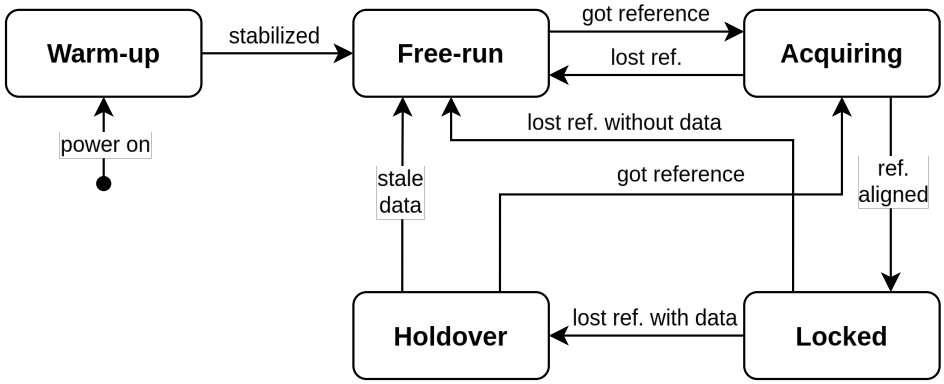
Free-run is a state entered after warm-up or loss of external reference, which guides the clock's operation. The clock's quality is purely dependent on its oscillator in this state. For instance, standalone atomic clocks based on caesium always run in this mode;

Acquiring state is entered when the clock gets a hold of an external reference to which it tries to syntonize and possibly phase align. The output of the clock is typically turned off in this state. This is called clock *squelching*;

Locked state is entered after the clock's control circuitry decides the alignment is within a specified range. The range is specified based on the clock's properties and the implementor's target application. Changes are made constantly to minimize the drift from the reference. If abrupt change is encountered, the clock may return to the acquiring state or possibly to holdover or free-run states based on how long it kept the reference locked;

Holdover state is entered when the clock loses the reference but is kept locked for a defined period. After this period, the clock can replay its historical behaviour against the reference to guide its oscillator. Because the clock's external environment constantly changes, the learned data slowly becomes stale, decreasing accuracy. The clock should return to the free-run state after the accuracy degrades too much. [12]

The clock's operator may also intervene in the clock's operation and force some of the states. Notably, the free-run state may be enforced. The same is true for the holdover state, regardless of the validity of the holdover data. Without the data, this state behaves the same as the free-run state. [12]



■ **Figure 3.3** State diagram of a clock. The diagram depicts the clock states based on the availability of an external reference and learned holdover data. The clock's operator may also force the free-run and holdover states. [12]

3.4.1 Osciallators

In modern electronics, the oscillator used is typically some form of a crystal oscillator (XO). Such oscillators are commonly made using a quartz crystal, an instance of a piezoelectric crystal. This means the crystal oscillates at a specific frequency when an electric current is applied. This frequency can be influenced by the shape the crystal is cut into and by the properties of the crystal used. However, the crystal's properties are furthermore affected by external factors such as humidity and temperature. This is called the crystal's *sensitivity*, which also alters as the crystal ages. [12]

To overcome the crystal's sensitivity, a variation of XO called the voltage-controlled crystal oscillator (VCXO) may be used. Its frequency is changed by applying changes to the supplied voltage. Nevertheless, the critical factor influencing XOs is the temperature of the environment. [12] Two types of XOs appeared to overcome this:

Temperature-compensated crystal oscillator (TCXO) integrates a circuit which applies correction voltage based on the ambient temperature. This is a voltage controlled solution;

Oven-controlled crystal oscillator (OCXO) encloses the crystal in an oven, which tries to keep the crystal's temperature constant, above ambient temperature. This is a temperature controlled solution. [12]

Both of these types are used in modern electronics requiring precise oscillators. The decision must be made based on the application's stability and cost requirements. At around three times the cost, OCXO can be 10 to 100 times more stable than TCXO. [12]

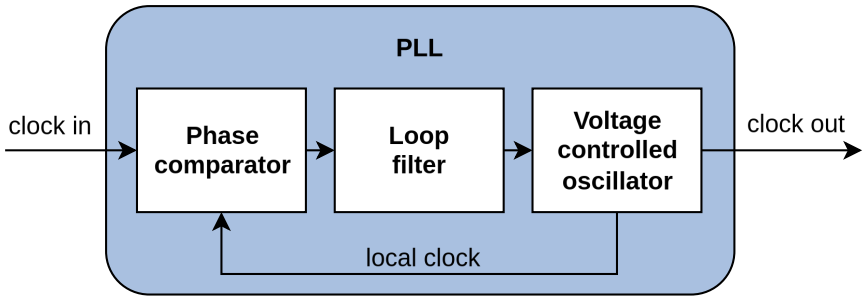
3.4.2 Phase-Locked Loops

A *phase-locked loop* (PLL) is a circuit generating a clock signal syntonized and phase-aligned to an input clock signal. Consequently, it is used as a base for all clocks which want to provide phase-aligned output. [12] PLL architecture is illustrated in Figure 3.4. It consists of three core blocks:

Phase comparator is a block which generates correction voltage based on the phase difference of the two clock signals;

Loop filter is a detection circuit which tries to filter noise, wander and jitter, from the input clock signal supplied as voltage by the phase comparator. This block is needed because the input signal accumulates noise during its physical transmission through the clocking network;

Voltage-controlled oscillator (VCO) is any oscillator controllable by correction voltage. It must supply its nominal, free-run, frequency f_0 when *nominal control voltage* is supplied. [12]



■ **Figure 3.4** PLL block diagram. The diagram depicts the clock states based on the availability of an external reference and learned holdover data. The clock's operator may also force the free-run and holdover states. [12]

The PLL starts with the nominal control signal applied to the VCO to provide its free-run frequency. The input signal phase fed into the phase comparator is then compared to the VCO output signal phase, and correction voltage is generated to increase or decrease the VCO frequency. The loop filter removes the noise on the correction voltage and then applies it to the VCO. This process constantly adjusts the voltage until small enough changes put the PLL into a locked state. This process runs while an input signal is detected. The continuous tracking of changes in both the input and VCO output keeps these signals tightly aligned. It enables the PLL to align itself with the input signal's long-term average while ignoring short-term changes. [12]

Similarly to clocks, the PLL is said to be in the *transient* state when it is not locked to the input reference and in the *steady* state when it is locked. The process of becoming locked is called *tracking*. [12] Three frequency ranges specified in ppm or ppb relative to the f_0 govern these states as specified by the ITU-T-REC-G.810:

Hold-in range is defined as "the largest offset between a slave clock's reference frequency and a specified nominal frequency, within which the slave clock maintains lock as the frequency varies arbitrarily slowly over the frequency range" [14]. The PLL can stay locked if the frequency difference between f_0 and input moves inside this range. It becomes transient when it reaches the edge of this range;

Pull-in range is defined as "the largest offset between a slave clock's reference frequency and a specified nominal frequency, within which the slave clock will achieve locked mode" [14]. That is, if the difference is in this range, the PLL goes from the transient state into the steady state after some number of cycles;

Pull-out range is defined as "the offset between a slave clock's reference frequency and a specified nominal frequency, within which the slave clock stays in the locked mode and outside of which the slave clock cannot maintain locked mode, irrespective of the rate of the frequency change" [14]. If the single-step applied difference is inside this range, the PLL stays in the steady state. Otherwise, it becomes transient. [12]

3.4.2.1 Filtering

The PLL loop filter typically uses a *low-pass* (high-cut) filter. Such a filter has a *cut-off* frequency defined. Frequencies higher than the cut-off frequency are *attenuated* (diminished). It splits the filter's band into the *stop band*, the attenuated frequencies, and the *pass band*, frequencies passed through the filter. Such a filter is illustrated in Figure 3.5. Hence, the PLL's bandwidth is the low-pass filter's pass band. Most PLLs allow this band to be configured based on the application's needs. [12]

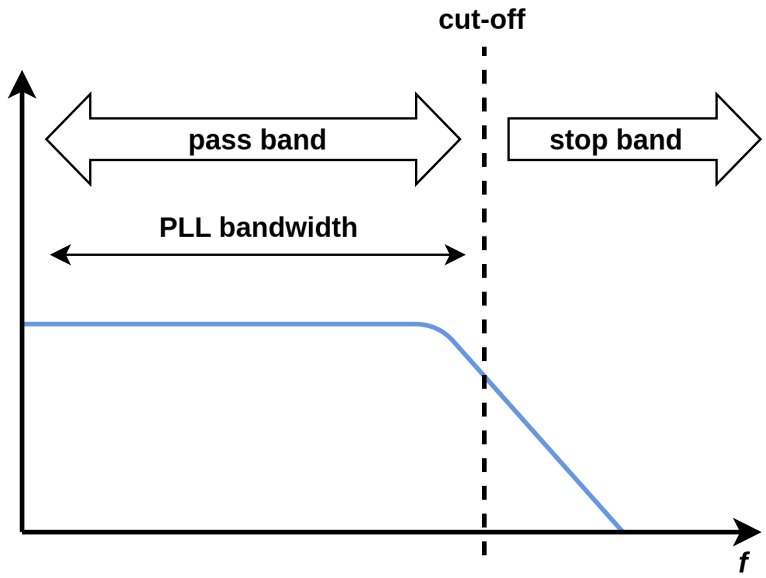


Figure 3.5 Band diagram of a PLL low-pass filter. The horizontal axis represents the frequency, and the vertical axis represents the signal strength. Illustrated are the two bands split by the cut-off frequency. The PLL's bandwidth is equal to the pass band. [12]

Using a low-pass filter implies that wander will pass through the PLL loop filter because it is a low-rate phase variation, whereas jitter will get filtered. That is, if the wander rate is lower than the low-pass filter cut-off frequency, and the opposite is true for jitter. There are multiple reasons for this decision in PLLs. [12]

Firstly, wander may reach microhertz frequencies. Its duration should always be measured for at least hours or better for days during the clocking testing. Implementing low-pass filters with such capability is impractical and not cost effective. [12]

Secondly, the PLL reacts to the combination of the input clock and the VCO output. If the low-pass filter bandwidth, in a corner case, would be zero, then only the clock of the local oscillator would be output by the PLL. This is against the point of a PLL, which is to have some reference to which it aligns. Also, this implies that an outstanding local oscillator is needed, which in turn furthermore pushes the cost up. [12]

Last, the smaller the pass band, the smaller changes can be made to the local VCO. This causes the PLL to take more time to reach the steady state. [12]

3.4.3 Errors

Even though a local clock is declared frequency and phase synchronized to some reference, the final output signal still carries errors. This is due to the oscillator quality and its enclosing environment. These errors are visible as *phase error*, which measures the difference in significant instants between the two signals. The unit used in networking is generally some unit of time, such as microseconds or nanoseconds. [12]

The core measurement is the *time error function*. It is the significant instant difference of the measured clock and the reference at time t, defined as:

$$x(t) := T(t) - T_{ref}(t) \tag{3.4}$$

where $x(t)$ is the time error function, $T(t)$ is the time of the measured clock and $T_{ref}(t)$ the time of the reference clock. [14]

If the measured clock arrives later than the reference, the resulting value will be negative and thus referred to as the negative time error. The opposite case, where the measured clock leads the reference, is the positive time error. [12]

Practically, attaining the continuous knowledge of the $x(t)$ function is impossible. A sequence of equally spaced samples thus substitutes it:

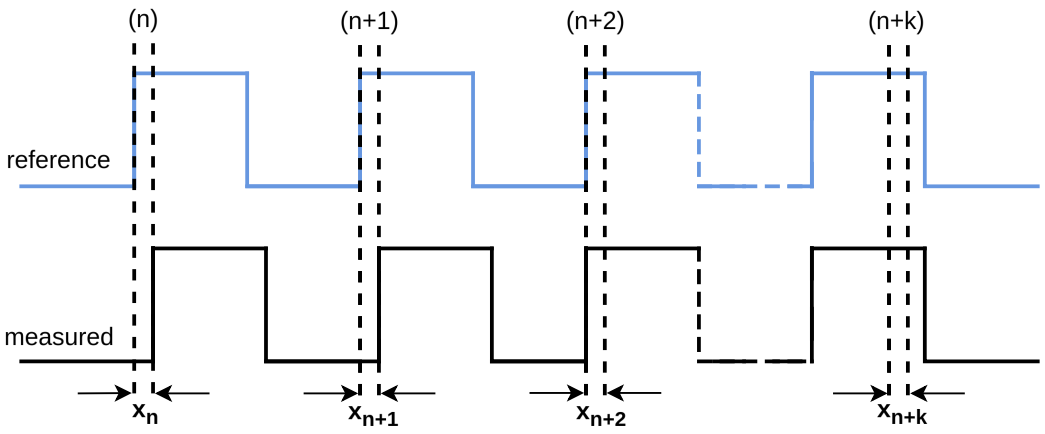
$$x_i = x(t_0 + i\tau_0) \tag{3.5}$$

where t_0 is the base time and τ_0 is the *sampling interval*. These samples are called the *time error* (TE). The sampling is illustrated in Figure 3.6. [14]

While the TE measures the difference at one instantaneous point in time, the *time interval error* (TIE) measures this change over an *observation interval* (τ). The interval duration is fixed at the measurement start. TIE thus measures the accumulated error over τ. TIE captures the signal jitter because TE measures the short-term variation in the clocking signal. As with TE, TIE is, in practice, measured using sample points. [12] Despite, it is defined as the *time interval error function*: [14]

$$TIE(t;\tau) := [T(t+\tau) - T(t)] - [T_{ref}(t+\tau) - T_{ref}(t)] = x(t+\tau) - x(t) \tag{3.6}$$

Two example observations present the usage of TIE. If TIE starts at a high value at the beginning of the measurement and then converges towards zero, this may indicate that the clock's PLL reached the steady state after the measurement started. The second example is the TIE increasing over the whole measurement duration. This occurrence indicates the clock has some frequency offset against its reference. [12]



■ **Figure 3.6** Time error and time interval error sampling. Illustrated are k sampling cycles as the observation interval. At each point n, the x_i illustrates the TE of that instant. The TIE for the first observation interval is calculated as $x_{n+k} - x_n$. [12]

The TE samples may be plotted, as in Figure 3.7. Three key properties may be extrapolated from this data:

Maximum absolute time error ($\max|TE|$) represents the largest difference of the two clocks. It is a single positive value because all the samples are taken as an absolute value;

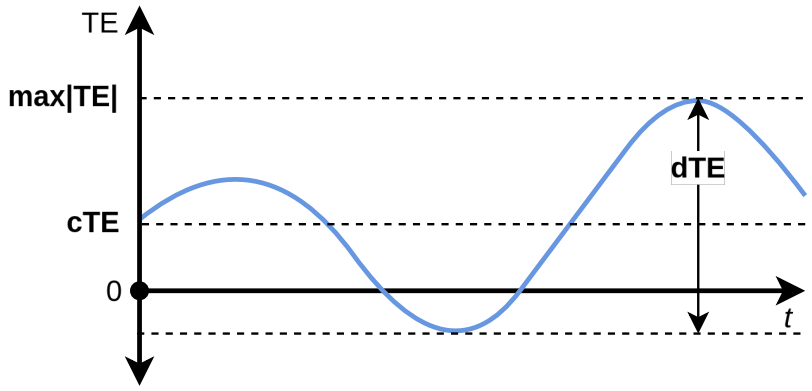
Constant time error (cTE) is the mean average of the sampled values from the beginning. It may be calculated over a specific duration from the beginning or the whole measurement duration. In the second case, it represents the average offset of the measured clock from the reference and is consequently a measure of its accuracy;

Dynamic time error (dTE) is the TE variation over a time interval. If this time interval is long enough, dTE represents wander in the clocking signal. It must be analyzed using the *maximum time interval error* (MTIE) and *time deviation* (TDEV). dTE is a measure of the stability of the clock. [12]

3.4.3.1 Maximum time interval error

The MTIE is defined as "the maximum peak-to-peak delay variation of a given timing signal with respect to an ideal timing signal within an observation time ($\tau = n\tau_0$) for all observation times of that length within the measurement period (T)" [14].

TIE measures the difference between the TE at the observation interval's start and end. This may not capture the maximum TE variation over this interval. MTIE requires finding the maximum and minimum TEs and capturing their difference. Additionally, its value is maximum over all observation intervals. Therefore, it never decreases. For example, if the MTIE over three-second periods is 20 ns, its value will never decrease



■ **Figure 3.7** Time error sample plot. Illustrated are the plotted TE samples, which are shown in blue. The cTE, max|TE| and dTE are also depicted. Note that this is only an illustration which is not based on real data and is not in scale. [12]

for periods longer than three seconds. It thus enables the identification of a frequency shift in the measured clock. [12]

The following formula is used for the mathematical estimation of MTIE:

$$MTIE(n\tau_0) \cong \max_{1 \le k \le N-n} \left[\max_{k \le i \le k+n} x_i - \max_{k \le i \le k+n} x_i \right] \quad (3.7)$$
$$\text{where } n = 1, 2, \ldots, N-1$$

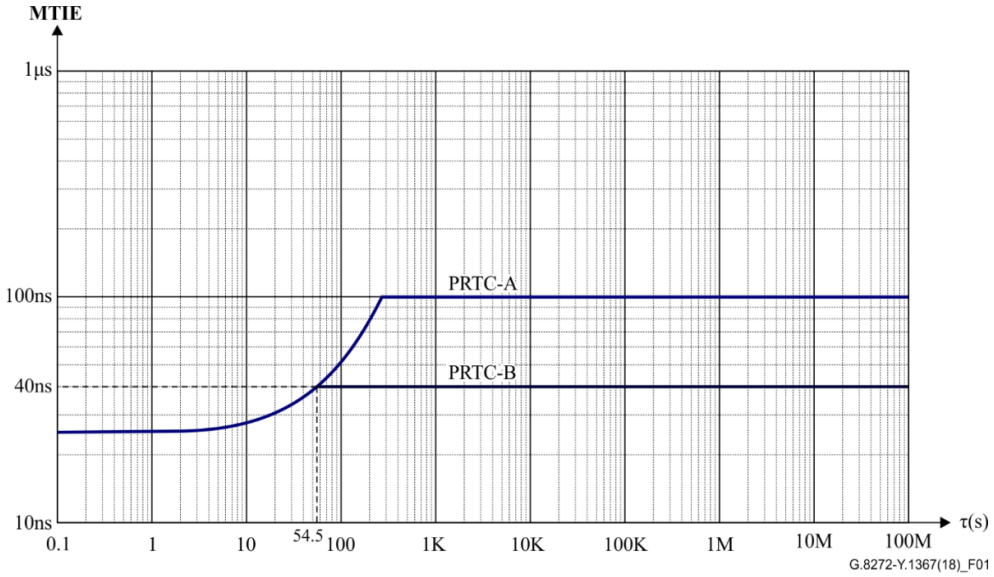
where $n\tau_0$ is the observation time τ with N samples of such duration over the whole measurement period T. [14]

If the MTIE values recorded are plotted, multiple essential observations can be made. If the graph is flat, no new maximum variations were recorded. New maximum variations are recorded if the graph diverges from zero to higher values. Last, if it increases linearly, the clock is not locked to the reference. The ITU defines so-called MTIE *masks*, which are line plots under which the measured clock's MTIE plot must remain for a specified measurement duration to be classified as a clock of some quality. Example masks for primary reference time clocks (PRTC) are illustrated in Figure 3.8. [12]

3.4.3.2 Time deviation

The definition of TDEV is "a measure of the expected time variation of a signal as a function of integration time. TDEV can also provide information about the spectral content of the phase (or time) noise of a signal" [14].

While MTIE observes the most significant phase differences, TDEV is a metric which measures how frequently these occur and how stable they are. In other words, TDEV measures the randomness of the clock's behaviour. [12]



■ **Figure 3.8** ITU-T-G.8272 PRTC MTIE masks. Illustrated are masks for PRTC class A and PRTC class B clocks. The flat line suggests that no new maximum variations should occur after some time. The MTIE for PRTC-A clocks is 100 ns and 40 ns for PRTC-B clocks. [16]

A statistical representation of such system instability may be provided by variance or standard deviation. However, it has been shown that the calculation of these measures does not converge for clocks. TDEV is thus needed. [12]

Mathematical estimation of TDEV based on a sequence of TE samples is done using the following formula:

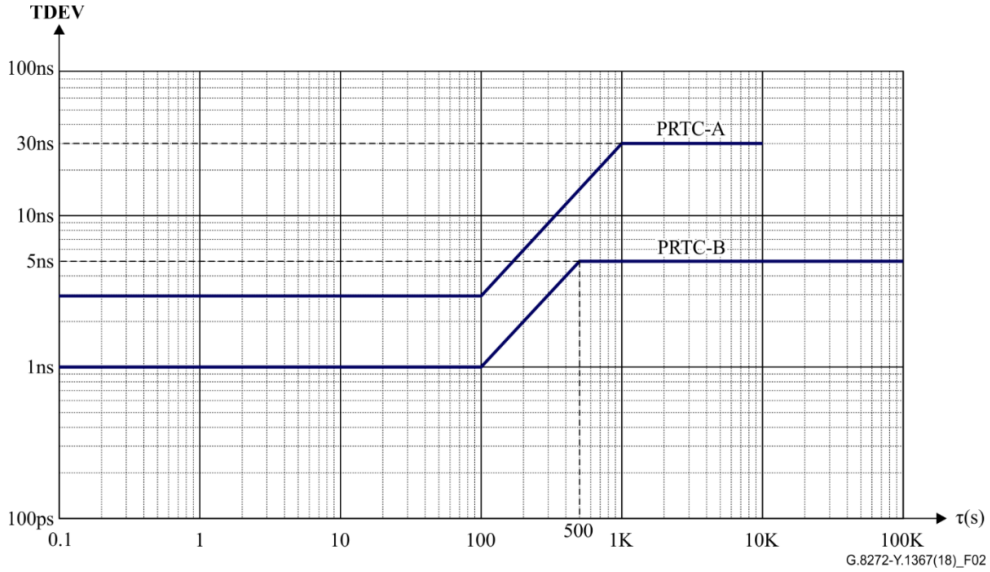
$$TDEV(n\tau_0) \cong \sqrt{\frac{1}{6n^2(N-3n+1)} \sum_{j=1}^{N-3n+1} \left[\sum_{i=j}^{n+j+1} (x_{i+2n} - 2x_{i+n} + x_i) \right]^2} \qquad (3.8)$$

$$\text{where } n = 1, \ldots, \left\lfloor \frac{N}{3} \right\rfloor$$

where N is the number of TE samples x_i and τ is the integration time. The integration time equals to $n\tau_0$ where n is the number of sampling intervals within τ and τ_0 is the TE sampling interval. [14]

TDEV can be imagined as the root mean square (RMS) of the clock's noise. RMS is calculated by taking the square root of the mean average of the square values of the samples. An essential property of RMS is that any extremes average out with enough data. This is the same with TDEV. It enables the filtering of rare occurrences, such as the sun starting to shine on the clock's enclosure and increasing its temperature, thus changing how the oscillator must be compensated. [12] The ITU T-G.811 recommends measuring TDEV for 12 times the duration of the maximum observation interval [17].

MTIE and TDEV are two fundamental metrics used to analyze the performance of clocks. MTIE helps to detect frequency offset and outliers in the TIE samples, whereas TDEV helps to average rare outliers and characterize the clock's wander. [12]



■ **Figure 3.9** ITU-T-G.8272 PRTC TDEV masks. Illustrated are masks for PRTC class A and PRTC class B clocks. The TDEV for PRTC-A clocks is 30 ns and 5 ns for PRTC-B clocks. [18]

3.5 Synchronous networks

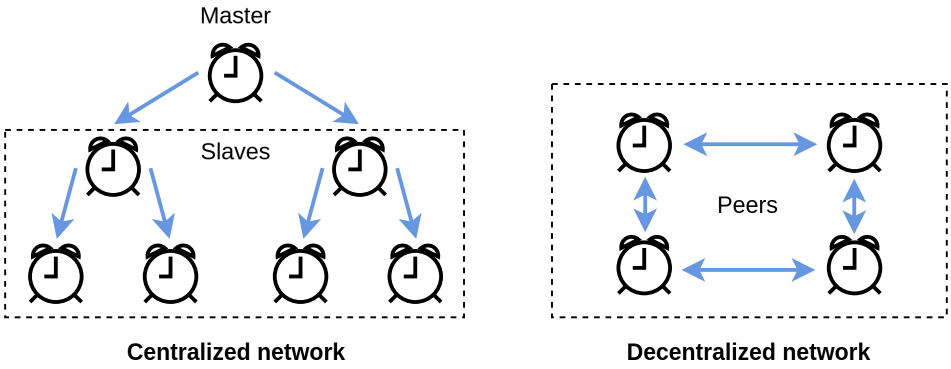
In asynchronous networks, nodes maintain their independent clock. Asynchronous technologies typically use some data frame delimiter. Ethernet, as an example, uses the frame preamble (PRE) to align the receiver (RX) frequency to that of the transmitter (TX) using the physical signaling sublayer (PLS). During Ethernet link negotiation, one side is selected as the master, and the other is selected as the link slave. The master uses its local free-runing clock frequency to send data. The slave recovers the master frequency using the PLS circuitry and uses it to send data back. There stands no centralized concept of clocking. [12]

However, this is not true with synchronous networks with a common clock reference. The critical fact is that this may convey frequency, phase, ToD, or any combination of the named. Devices which provide a reference signal of all three are the PRTCs. [12]

Synchronous networks are divided into two groups based on the control method:

Centralized networks are designed in a tree-like master-slave hierarchy. This provides one grandmaster clock as the absolute clocking reference for the whole node tree;

Decentralized networks transfer timing information between all neighbouring peers and try to determine the best quality one and let all the peer clocks contribute. [12]



■ **Figure 3.10** Synchronous networks control methods. On the left are illustrated slave clocks synchronizing to one master in a centralized network. On the right are depicted cooperating peers in a decentralized network. [12]

Furthermore, synchronous networks are divided based on the delivery method:

Physical synchronization networks use a physical clock signal carrier. An example may be the legacy Synchronous Digital Hierarchy (SDH) technology, which creates synchronous optical ring networks for periodic transmission of jumbo frames utilizing time-division multiplex (TDM);

Packet based synchronization exchanges time representation through timestamps carried using packets. For example, in the case of PTP over Ethernet, packets may be delayed by waiting in a switch FIFO; thus, the time at which the timestamp is taken at the sender and receiver is critical. Consequently, packet-based synchronization is burdened with packet delay variation (PDV), which is the varying transmit time of the periodic timestamp packets. That is, the periodicity is not obeyed perfectly. The significant instant of such methods must be specified for phase alignment, which is commonly some defined part of the packet, such as the Ethernet start-of-frame (SOF) delimiter. [12]

3.5.1 Traceability

The clocking signal degrades in all of these scenarios as it is carried further from the primary source. Because the goal is to provide the most accurate representation of the primary clocking signal, the nodes need to understand the quality of the incoming signal and where it is coming from. *Traceability* solves this problem in synchronous networks. Generally, a type of clock quality message is exchanged between nodes, synchronization status message (SSM) as an example in the case of SDH. [12]

A common problem to these methods is a *timing loop*. A topology may be constructed, where a clock X gives time to Y, which gives time to Z, which gives time back to X. This may also happen when a signal to a master is lost, and this group of clocks

creates a loop between them. No synchronization errors are raised because the clocks are synchronized despite not being synchronized to the master's clock. Traceability may be used to solve this problem. [12]

3.6 Synchronous Ethernet

Ethernet never needed synchronization of its circuitry because it was designed to carry asynchronous data. As explored in the previous section, only frequency synchronization is done on a link basis, where one side is selected as the master and the other as the slave, which turns around the master's RX frequency on its TX path. [12]

This property of Ethernet drove the price of the hardware needed down and was, in turn, one of the causes for its rapid adoption. Service providers started migrating synchronous networks based on technologies such as SDH to Ethernet. Data transmission conversion from TDM to packets brought the need for migration of the timing network. Accordingly, the distribution of frequency over Ethernet was needed as in SDH. [12]

The result of this requirement is *Synchronous Ethernet* (SyncE). Slave clocks extract timing from the frame's preamble carried by the Ethernet line coding. This enables the use of SyncE on optical and copper-based links while not being impacted by high packet loads. Ethernet links of 10 Mb/s bandwidth are exempt from this technology because *idle periods* occur when there is no data to transfer. During these periods, nothing is sent on the physical layer; thus, the frequency cannot be recovered. [12]

3.6.1 Ethernet equipment clocks

Specialized hardware is needed to support SyncE. The slave must incorporate a PLL, which locks onto the master frequency. Timing traces must also be incorporated to transfer the timing signals between the Ethernet and timing circuitry. The recovered clock is used to drive all the Ethernet interfaces to provide the clock signal to further downstream nodes. Such devices are called *Ethernet equipment clocks* (EEC). [12]

3.6.2 Ethernet Synchronization Messaging Channel

SyncE achieves traceability utilizing the *Ethernet Synchronization Messaging Channel* (ESMC). EECs communicate on this channel using the Ethernet overhead function called the *slow protocol*. For example, the Link Aggregation Control Protocol (LACP) is also an Ethernet overhead function. *Information* protocol data units (PDU) carry quality indicators periodically, whereas *event* PDUs are generated upon events such as a quality change or a link failure. [12]

3.7 Precison Time Protocol

The IEEE 1588 Precision Time Protocol provides packet-based clock synchronization. It can synchronize frequency, phase and ToD. It is extensible using so-called *profiles* while default profiles permit the installation of minimal systems without user intervention. With minimal computing and network resources, sub-microsecond accuracy is achievable. In properly designed networks, see Section 3.8 for an example, sub-nanosecond accuracy can be achieved. [19] Note that the protocol uses the term *message* for packets.

This accuracy is achieved by forming master-slave relationships in the network, where masters provide clock to slaves, utilizing support from hardware clocks for capturing packet timestamps. The clock of all nodes in the network is traceable to the grandmaster (GM) clock, which acts as a source of truth for time in the network. The GM typically sources itself with a clock from a reference such as a PRTC or the GNSS. [12]

3.7.1 Network Time Protocol

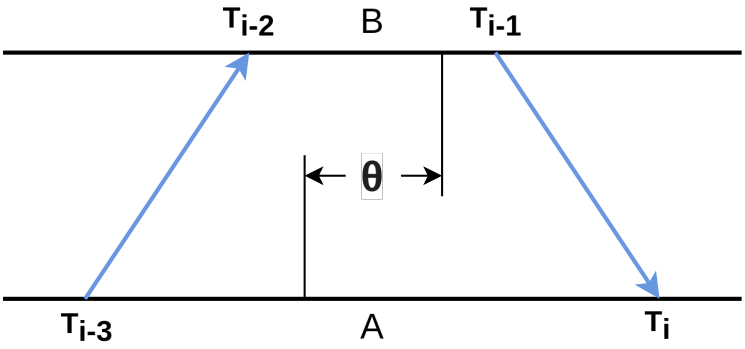
The first version of PTP from 2002 was based on multicast transport with the time-to-live (TTL) field of packets set to zero. It was designed for clock synchronization of nodes in close proximity. This transport limitation made it unsuitable for use in wide area networks (WAN) such as the Internet. [12]

On the other hand, the Network Time Protocol (NTP) was invented for this purpose in the 20th century. The goal it was designed to handle was to synchronize servers running legal and financial transactions, transportation and other applications involving distributed resources over the Internet. [20]

NTP organizes servers as a hierarchical structure, in which every server's level is given by its *stratum* number, to which all clients connect [7]. Its synchronization mechanism is based on a *two-way* exchange of messages between the peers depicted in Figure 3.11. The offset of the client from the server θ and the network roundtrip delay δ are periodically recalculated. [20] These two values are passed as an input into the clock discipline algorithm, a hybrid phase/frequency-locked loop software implementation [21]. The result of this algorithm is the adjustment of the local clock. This is done on a pool of servers by the client from which the peer-selection algorithm chooses the source *truechimer*. Servers determined to be providing inaccurate time are discarded as falsetickers. [7]

Employing this software-based method, NTP can reach millisecond accuracy over the Internet [12], sub-millisecond accuracy utilizing a PPS signal [22]. However, two difficulties arise when calculating the critical packet-based synchronization parameter, the network delay [9].

Firstly, the time capture (T_i) of the event messages is a software event providing accuracy of tens of microseconds at best. PTP standardizes the use of physical layer timestamps, providing nanosecond accuracy. [9]



■ **Figure 3.11** NTP offset and delay measurement. Depicted are server A and peer B. Let $a = T_{i-2} - T_{i-3}$ and $b = T_{i-1} - T_i$. Then, the clock offset $\theta_i = (a+b)/2$ and the roundtrip delay $\delta_i = a - b$ of B relative to A at time T_i. NTP messages always carry the last three timestamps, and the fourth (T_i) is captured upon reception of each message. [20]

Secondly, the delay is not constant. NTP tries to overcome this by taking many samples and applying its filtering algorithms. Since the 2008 revision, PTP has introduced the concept of a transparent clock, which dynamically compensates for network buffering delays by adjusting the passing messages. [9] This revision removed the multicast limitation and allowed a broader adoption [12].

3.7.2 Entities

The PTP standard defines multiple key terms. The physical device utilizing the protocol is the PTP *node*. It may have one or more physical network interfaces. On nodes run PTP *instances* of the protocol. Each instance has its *local clock* (LC) assigned and its *local PTP clock* (LPC) assigned. The LC is a physical clock used in the protocol's operation. It may be syntonized to a reference clock. The LPC provides a local estimate of the GM time. It may be a physical or a mathematical clock. LC and LPC are allowed to be the same clock. The node may provide a dedicated LC for each instance, or they may share a common one. Each instance has its own LPC. The relationships between these entities are illustrated in Figure 3.12. [19]

Nodes in PTP belong to one or more domains. Instances belong to exactly one domain. A *domain* is a network of instances where one GM is traceable as the time source. Thus, the domains are logically separated and do not communicate. This means that numerous domains may coexist on the same physical network. [12]

Each domain is identified by a number from 0 to 255. Values greater or equal to 128 are reserved. The domain 0 is called the *default domain*. The profiles define the used domains and their numbers. Usually, only a small count is used, for instance, two domains in telecommunication profiles. [12]

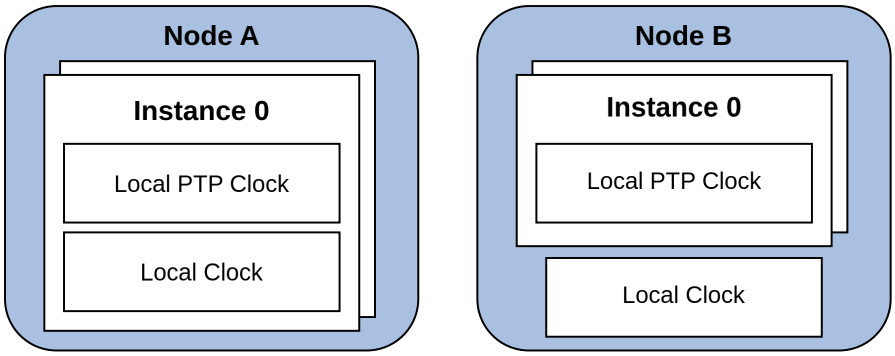


Figure 3.12 PTP relationship between nodes, instances and clocks. Depicted is node A which provides unique local and local PTP clocks for each instance. Node B illustrates a case where all instances share a common local clock. [19]

3.7.3 Ports

A PTP *port* is a logical access point of an instance into a domain [19]. One instance may have multiple PTP ports. A single physical interface of a node may have multiple PTP ports associated with it. It is a purely virtual concept. Each port has exactly one domain and one protocol version associated with it. Consequently, all connected ports belong to the same PTP domain. The relationship between domains, nodes and ports is illustrated in Figure 3.13. [12]

Each port is identified by a structure called `portIdentity`. It is formed from a field `clockIdentity`, an eight-byte wide identifier of the port's instance and a field `portNumber`, the port's 16-bit unsigned identifier starting at 1. These fields may be used for tie-braking when selecting the best master clock. [12] The `clockIdentity` field may get assigned the clock's network unique identifier (NUI)-64, or NUI-48 and implementor defined 16-bit remainder. The implementor must ensure his assignment provides unique identifier within the domain. NUI is defined in the IEEE 802. [19]

Ports transition between three states:

MASTER when the port is a source of time on its communication path;

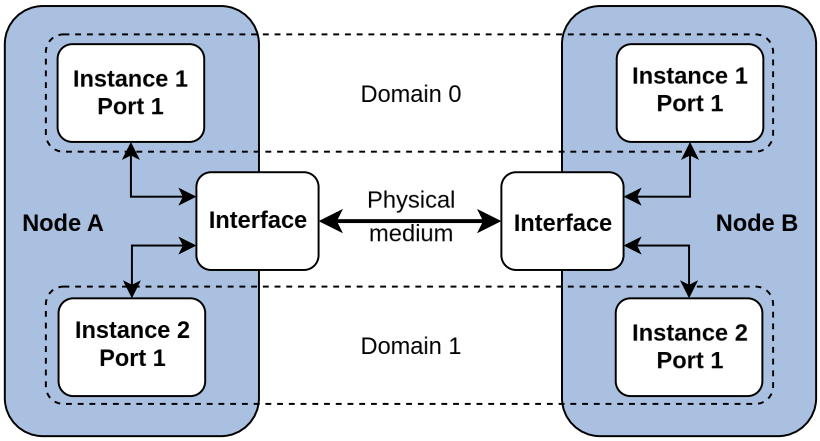
SLAVE when the port synchronizes to a master on its communication path;

PASSIVE when the port is neither a source of time nor synchronizes to a master. [19]

3.7.4 Messages

PTP defines two types of messages:

Event messages are sensitive to delay. The slave's clock synchronization logic is sensitive to the delay accumulated during transmission. For these messages, time is recorded at their transmission and on their reception on the message target;



■ **Figure 3.13** PTP relationship between domains, nodes and ports. Depicted are two PTP nodes, each with one physical network interface and two instances with one port mapped to the interface. Each instance and port is mapped to a different PTP domain, while both nodes belong to both PTP domains. [19]

General messages are not sensitive to delay because they carry control and management data. They may still carry the captured time for some event messages, but their delivery is not time-sensitive. [12]

The complete list of messages defined for both types is provided in Table 3.1. The use and meaning of these messages is examined further.

■ **Table 3.1** PTP event and general message list. All defined messages are listed. [12]

Event	General
Sync	Follow_Up
Delay_Req	Delay_Resp
Pdelay_Req	Pdelay_Resp_Follow_Up
Pdelay_Resp	Announce
	Signaling
	Management

The protocol defines multiple transport methods for its messages between ports in its annexes. It is the profile that decides the methods that are allowed and used. The first version of the protocol from the year 2002, PTPv1, utilised multicast transport. The following revision from 2008, PTPv2, added unicast support for the User Datagram Protocol (UDP) over the Internet Protocol (IP). [12] The UDP port 319 is assigned for PTP event messages, and port 320 for general messages in the *Service Name and Transport Protocol Port Number Registry* of the Internet Assigned Numbers Authority (IANA) [23]. The complete list of supported transports in the 2019 revision of the protocol is provided in Table 3.2.

Messages are exchanged at a *rate* specified per message [12]. The rate is expressed as a $\log_2(\texttt{seconds})$ with possible values ranging from -128 to 127 [19]. For example, that is -3 for one message every 1/8 of a second. The assigned rates are defined in the profile and may sometimes be adjusted on the node by the operator [12].

Table 3.2 PTP supported transport mechanisms. All the transport mechanism annexes defined in the 2019 protocol revision are listed. [19]

Transport	Annex
UDP over IP version 4	C
UDP over IP version 6	D
Ethernet	E
DeviceNET	F
ControlNET	G
PROFINET	H

3.7.5 Timestamps

The core of PTP lies in its timestamps. A *timestamp* is defined as a structure, which "represents a positive time with respect to the epoch" [19]. It consists of a 48-bit wide unsigned integer called `secondsField` representing seconds since the epoch and a 32-bit unsigned integer called `nanosecondsField` counting the nanoseconds following the seconds. The nanosecond field is strictly less than 10^9. That is the number of nanoseconds present in one second. [19]

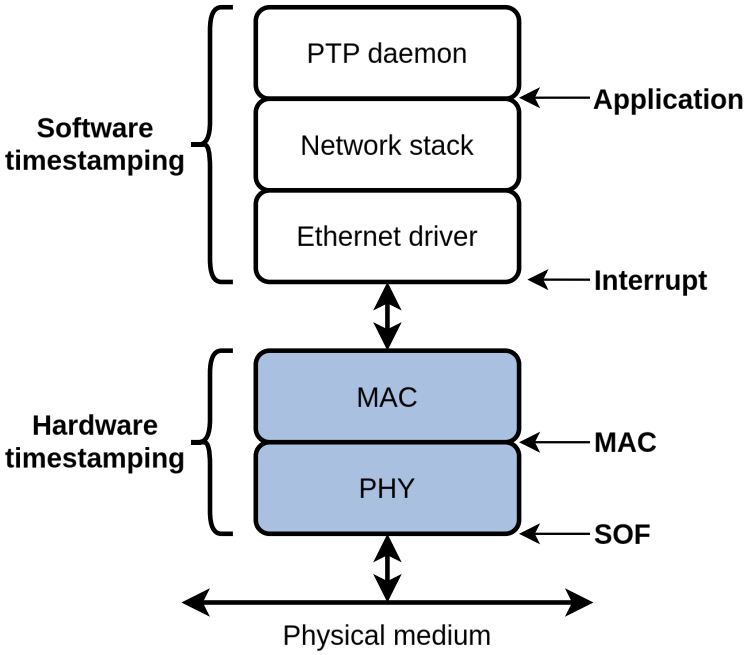
The protocol may use two types of timescales for its timestamps: PTP or arbitrary (ARB). The ARB timescale is implementation-specific, with the condition that it is a monotonic timescale. The PTP timescale epoch is defined as 1 January 1970 00:00:00 TAI. The offset to UTC may be thus calculated as the PTP time minus ten, the offset of TAI and UTC on 1 January 1972, the introduction of leap seconds, and minus the number of leap seconds since then. At the time of writing, that is the PTP time minus 37 seconds. [12]

Timestamps are generated for PTP event messages by the *timestamping clock*. This may be either the LC or the LPC. [19] It is paramount to the accuracy of the synchronized time to capture the timestamps with the best possible quality [7]. The timestamp capture point is defined as the SOF delimiter of the frame, but the transport layer annexes to the standard may specify otherwise [19]. The captured timestamps may be differentiated into two types:

Hardware timestamps are generated using dedicated circuitry. In the case of Ethernet, this may be done either by the PHY or the MAC;

Software timestamps are traditionally captured by an interrupt handling the frame reception. Because software may introduce non-deterministic processing delay, it increases the timestamping jitter and is thus avoided in applications targeting a high level of accuracy. [7]

In both cases, additional delay is introduced against the optimal timestamping point and must be thus corrected. An example on the hardware level is the delay in the PHY transferring the frame to the MAC. [7] This delay is often asymmetric for RX and TX paths and must be compensated for by the implementation [19]. The possible timestamp capture points are illustrated in Figure 3.14.



■ **Figure 3.14** Diagram of possible PTP timestamp capture points. Depicted are software and hardware-based points for capturing PTP event message timestamps. [7]

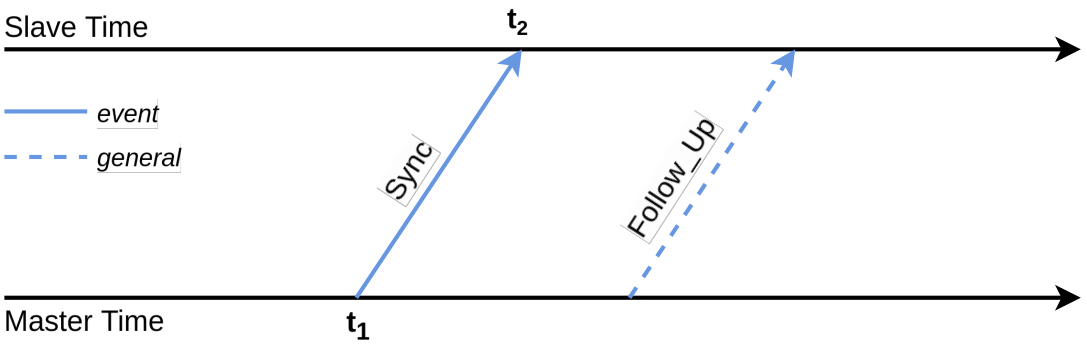
3.7.6 Synchronization mechanism

The PTP synchronization is based on a two-way message exchange mechanism similar to that of NTP. The slave must determine its offset from the master to adjust its time to where it should be. [7] This offset may be calculated by subtracting the master's time from the slave's time. In PTP, this is defined as: [19]

$$\texttt{offsetFromMaster} = T_{slave} - T_{master} \tag{3.9}$$

Negative result of this calculation means the time on the slave passes slower than on the master, whereas a positive value means the time passes faster on the slave [12].

PTP uses the `Sync` message periodically sent by the master to its slaves to announce its current time t_1. The slave captures its time t_2 at the reception of this message from the master. Note that the t_1 is the master's time when the message was sent. This means either the master's hardware must be capable of inserting the timestamp while actively sending the message, such clock is referred to as a *one-step* clock, or the actual t_1 is sent in the `Follow_Up` message. This method is referred to as the *two-step* clock and is chosen by the master by setting the `twoStepFlag` field in the `Sync` message. The slave must be capable of handling both methods. This illustrates Figure 3.15. [12]



■ **Figure 3.15** PTP time synchronization. Depicted is the master sending its time t_1 in the `Sync` message in one-step mode and the slave capturing the reception timestamp t_2. Optional `Follow_Up` message carrying the master's t_1 in the case of two-step mode is also illustrated. [19]

Because the slave calculates the offset based on these two timestamps, delivering the `Sync` message without unnecessary delay is of utmost priority because its reception is timestamped as the t_2. On the other hand, the `Follow_Up` message is not of the same criticality because it just carries the master's t_1 and is not itself timestamped. [12]

However, as in all packet-based time synchronization protocols, the transit time of the `Sync` message through the network must be considered. Without subtracting it, the offset includes the transmit time as an error. This term is called the *one-way propagation delay* in PTP. The value subtracted in the offset calculation is called the `meanDelay` because it is the average of the roundtrip delay. [12] Two mechanisms for measuring it are defined: delay request-response and peer-to-peer. The offset equation is thus: [19]

$$\texttt{offsetFromMaster} = t_2 - t_1 - \texttt{meanDelay} \tag{3.10}$$

The protocol introduces a third mechanism to improve the accuracy of the calculated offset called the `correctionField` carried in PTP messages. It is a 64-bit wide unsigned integer field of nanoseconds multiplied by 2^{16}. Consequently, it allows the representation of subnanosecond precision, with 48 bits of nanoseconds and 16 bits of nanosecond fraction. [12] Its use is discussed in the following sections.

In the case of the offset, the complete equation, including the `correctionField`, is:

$$\texttt{offsetFromMaster} = t_2 - t_1 - \texttt{meanDelay} - \texttt{correctionField} \tag{3.11}$$

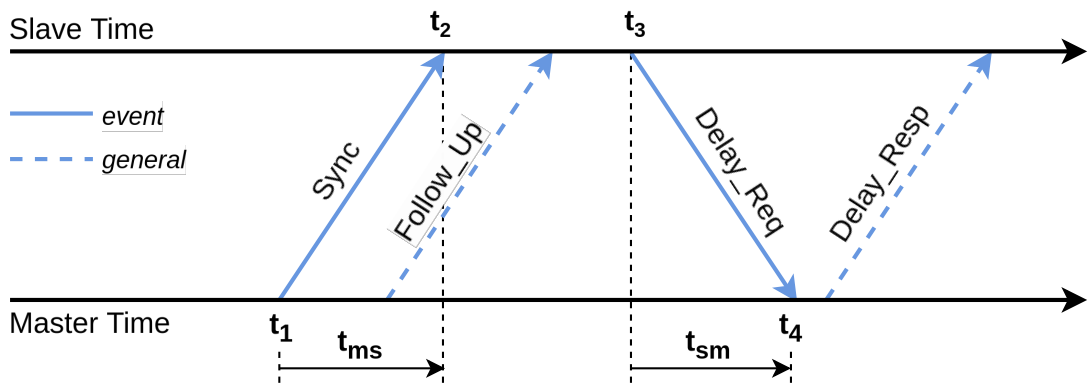
The subtracted value is this field of the `Sync` message in the one-step mode. In the case of a two-step operation, this field of the `Follow_Up` message is subtracted in addition to the one in the `Sync` message. [19]

The critical fact is that the standard does not define how the resulting offset should be applied to the slave's clock. It is up to the implementation to adequately align the slave. This may be done, for example, by forcefully applying the offset or changing the underlying oscillator frequency for a while to smooth out the change and prevent any extreme jumps in time. [24]

3.7.7 Delay request-response mechanism

The delay request-response mechanism, also referred to as the end-to-end (E2E) mechanism, provides means to measure the `meanPathDelay`, which is the representation of the `meanDelay` between two ports connected by any network topology. It is initiated periodically by the slave against its master. [7]

The slave issues the `Delay_Req` message to the master and notes its time t_3. This is an event message, same as the `Sync` message, so it must be sent without delay. The master timestamps the reception of this message as t_4 and sends it back in the general `Delay_Resp` message. After the response reception, the slave calculates the one-way delay utilizing timestamps t_1 and t_2 from the last `Sync` message. The full message exchange is illustrated in Figure 3.16. [19]



■ **Figure 3.16** PTP delay request-response mechanism. Depicted is the measurement of the `meanPathDelay`, which is the average of the one-way propagation delays of the `Sync` and `Delay_Req` messages, assuming a symmetric connection delay. [19]

The slave first calculates the one-way propagation delay for both message paths, that is, master to slave and slave to master, as:

$$t_{ms} = t_2 - t_1 \tag{3.12}$$

$$t_{sm} = t_4 - t_3 \tag{3.13}$$

and then the `meanPathDelay` as an average of these two values because the connection delay is assumed to be symmetric:

$$\texttt{meanPathDelay} = \frac{t_{ms} + t_{sm} - \texttt{correctionField}}{2} \tag{3.14}$$

In the case of a one-step operation, the `correctionField` is equal to the sum of this field in the `Sync` and `Delay_Resp` messages. In the two-step mode, it is the same sum while additionally adding this field in the `Follow_Up` message. [19]

3.7.8 Peer-to-peer delay mechanism

In a peer-to-peer (P2P) delay mechanism, the `meanDelay` is measured between two neighbouring ports and referred to as the `meanLinkDelay`. Ports are not allowed to communicate with more than one peer. Both sides initiate this mechanism to measure the delay. The `Delay_Req` and `Delay_Resp` messages are dropped. Thus, the E2E `meanPathDelay` cannot be measured between the slave and the master. Consequently, one domain cannot combine the E2E and P2P mechanisms. [12]

The P2P mechanism requires the aforementioned transparent clocks in the network to adjust the propagation delay on a link basis in the `correctionField`. The advantage is that a delay correction occurs immediately upon routing changes. [7]

The mechanism is similar to how the E2E mechanism measures the delay based on the `Sync` and `Delay` timestamps. The requestor sends the `Pdelay_Req` event message noting the transmit timestamp t_1. The responder timestamps the reception as t_2 and sends it in the event message `Pdelay_Resp` with t_3, the transmit timestamp. The requester timestamps the reception of this message as t_4. As with the `Sync` message from a master, the responder may not be able to provide t_3 in the message, so a two-step `Pdelay_Resp_Follow_Up` may come later with the t_3. This complete message exchange is illustrated in Figure 3.17. [19]

Assuming a symmetric connection delay, the `meanLinkDelay` is calculated as:

$$\texttt{meanLinkDelay} = \frac{[(t_4 - t_1) + (t_2 - t_3)] - \texttt{correctionField}}{2} \tag{3.15}$$

As with the E2E mechanism, the measured one-way propagation delays must be corrected using the `correctionField` before the division. The subtracted value is this field

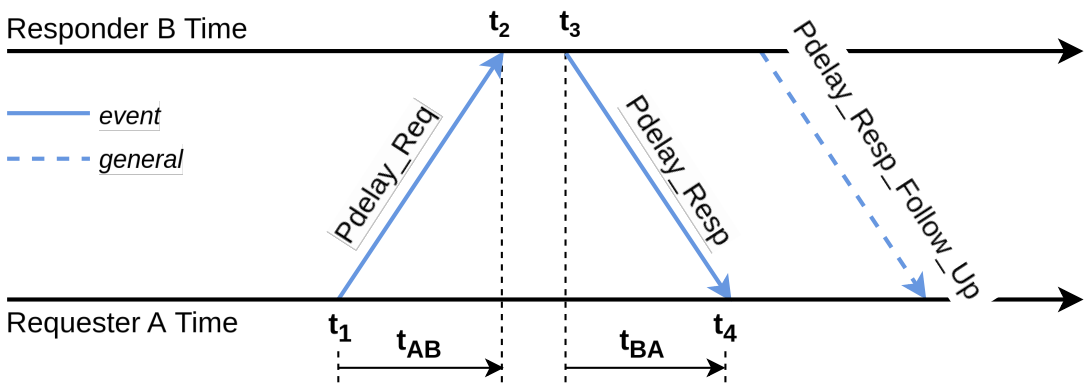


Figure 3.17 PTP peer-to-peer delay mechanism. Depicted is the measurement of the `meanLinkDelay` between two peers, which is the average of the one-way propagation delays of the `Pdelay_Req` and `Pdelay_Resp` messages, assuming a symmetric connection delay. [19]

of the `Pdelay_Resp` message in the one-step mode. In the case of a two-step operation, this field of the `Pdelay_Resp_Follow_Up` message is subtracted in addition to the one in the `Pdelay_Resp` message. [19]

3.7.9 Asymmetry correction

All the PTP timing equations assume the transmit time of messages is symmetric in both directions. In practice, this is not true. The transmission is permanently burdened with two types of asymmetry:

Static asymmetry is the part which never changes. It can be measured using specialized timing equipment or estimated with a certain degree of accuracy. For instance, long fiber runs may introduce a measurable impact of chromatic dispersion, the change of velocity of the electromagnetic wave through the fiber. Another example is a length mismatch between fiber pairs;

Dynamic asymmetry cannot be fully compensated for in practice because it constantly changes. An example is the message queuing in network equipment or the message target during high network load. [12]

The main difficulty is understanding which part of asymmetry is static and which is dynamic. PTP provides a means to compensate for the static asymmetry in all the calculations by providing a value for the `delayAsymmetry` field. [12] It is defined as:

$$t_{ms} = \texttt{meanPathDelay} + \texttt{delayAsymmetry} \tag{3.16}$$

$$t_{sm} = \texttt{meanPathDelay} - \texttt{delayAsymmetry} \tag{3.17}$$

or

$$t_{\text{resp-to-req}} = \texttt{meanLinkDelay} + \texttt{delayAsymmetry} \tag{3.18}$$

$$t_{\text{req-to-resp}} = \texttt{meanLinkDelay} - \texttt{delayAsymmetry} \tag{3.19}$$

That is, it is positive if the master-to-slave or responder-to-requester propagation time is longer than in the other direction. It is applied to the `correctionField` before the calculations are done. [19] It is out of the scope of this work to discuss how this is done by the protocol in all cases.

For direct connection of two PTP ports using certain bidirectional media, the value of `delayAsymmetry` is almost equal to the underlying physical medium asymmetry. An example is the 1000BASE-BX10 defined in IEEE 802.3. In such cases, the `delayAsymmetry` may be calculated as follows: [19]

$$\texttt{delayAsymmetry} = \texttt{constantAsymmetry} + \frac{\alpha}{\alpha + 2} \times \texttt{meanDelay} \tag{3.20}$$

The parameter α is called the `delayCoefficient`. It is the relative difference between the two transmission times defined as:

$$t_{ms} = (1 + \alpha) \times t_{sm} \tag{3.21}$$

$$t_{\text{resp-to-req}} = (1 + \alpha) \times t_{\text{req-to-resp}} \tag{3.22}$$

It is positive if the master-to-slave or responder-to-requester propagation time is longer than in the other direction. The `constantAsymmetry` value may be provided to adjust the `delayAsymmetry` calculation based on additional known static asymmetry. [19]

3.7.10 One-way mechanism

PTP may be used only for syntonization employing a *one-way* message exchange, the reception of the master's `Sync` messages by the slave. This is possible because, for syntonization, only the rate at which its time passes compared to its master must be known. [12] The slave captures the `Sync` events at a rate of `syncInterval` [19]. After receiving at least two such events, it may calculate the frequency difference as follows:

$$\frac{(t_{N2} - t_2) - (t_{N1} - t_1)}{t_{N1} - t_1} \tag{3.23}$$

where t_N denotes the timestamp captured for the N-th event. [24]

Utilizing this method may be slow depending on the `syncInterval` defined by the profiles. It may take up to an hour to reach syntonization. The slave must do this measurement over an extended period because it tries to detect minor frequency errors

in its local clock. Instead, a physical layer syntonization method may be employed because it provides almost instantaneous syntonization. This combination of PTP for phase and ToD synchronization and a physical layer syntonization is called *hybrid mode*. The most common example of this is implementing SyncE together with PTP. [12]

3.7.11 Node types

PTP defines five types of nodes: management nodes, ordinary clocks, boundary clocks, E2E transparent clocks, and P2P transparent clocks. [12]

3.7.11.1 Management node

Management nodes do not participate in the synchronization. They modify the protocol's configuration and state variables other nodes keep in *datasets*. This is achieved by dispatching `Management` messages, which carry elements encoded using a type-length-value (TLV) mechanism defined by the protocol. They are rarely used, and some profiles, such as the telecommunication ones, prohibit their use. Because of this, they will not be further discussed. [12]

3.7.11.2 Ordinary clock

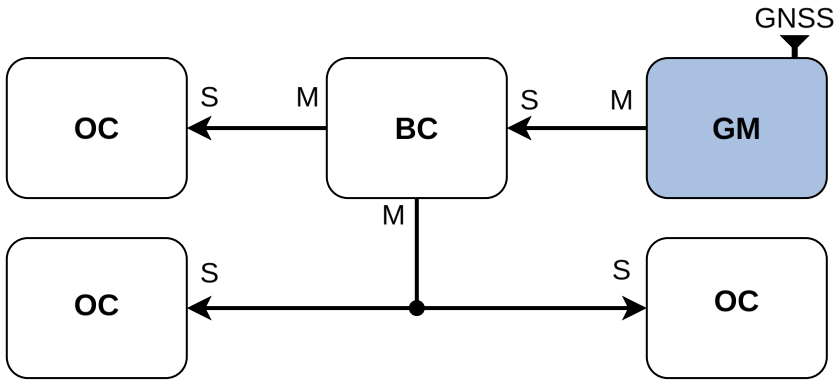
The *ordinary clock* (OC) is the simplest PTP clock node. It has a single port, which is either in the master or slave state. Consequently, it must lie at the end of the domain's hierarchy because it cannot further distribute the clock. Hence, OC is either a slave or the GM. That is because if it is not a slave, it must be a master, and if it is a master, it must have been chosen as the best one. That is because the quality of all other clocks came from this port and was decided to be worse than the port's quality. This assumes the operator did not disable this port or force its state. [12]

3.7.11.3 Boundary clock

Boundary clocks (BC) have at least two ports, which transition between the three possible states. They synchronize to some master and act as a master to other nodes. This is important because they are a point of resetting the error introduced by PDV. When the synchronization messages are received from their master, they are terminated. New synchronization message flow is started in the direction of their slaves. Simple hierarchy featuring OCs and BCs is illustrated in Figure 3.18. [12]

3.7.11.4 Transparent clock

Transparent clocks (TC) exist to adjust the timestamps of synchronization messages passing through them. Let us examine an Ethernet-based PTP application as an example. The propagation delay of the `Sync` message includes the time this message stayed



■ **Figure 3.18** PTP master-slave hierarchy. Illustrated is one grandmaster synchronized to the GNSS. One boundary clock as its slave is depicted as a master of three other ordinary clocks. [12]

inside each network bridge it passed through while waiting to be passed from one of its ports to some other. This directly delays the capture point of the t_2 timestamp by the slave, which may become problematic because slaves discard outlier offsets from their masters. Bridges acting as TCs insert this *residence time* into the passing messages so that the slaves can account for this. [12]

The residence time is calculated as follows:

$$\texttt{residenceTime} = \texttt{egressTimestamp} - \texttt{ingressTimestamp} \tag{3.24}$$

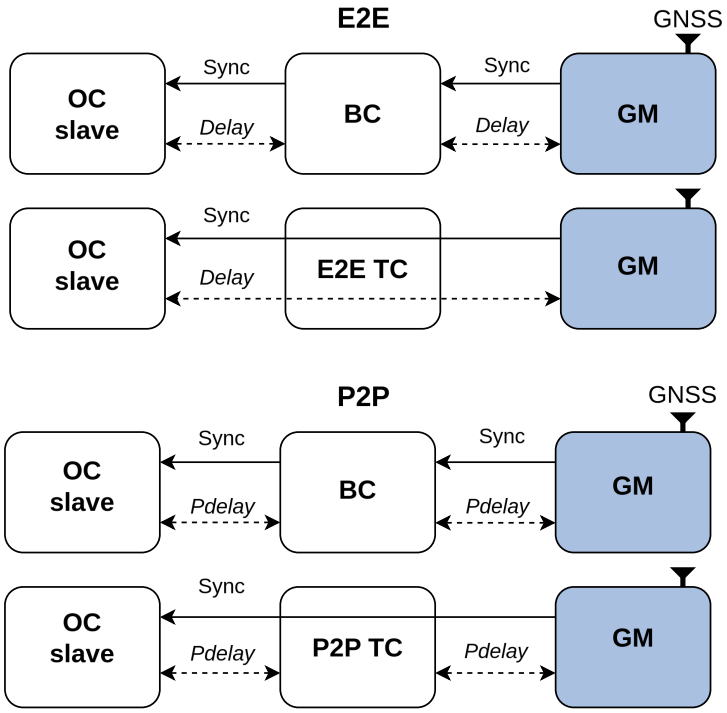
for the passing messages where the timestamps are the ones captured by the TC. As with the delay mechanisms, it depends on the hardware capabilities of the TC if it can insert the `residenceTime` into all the passing messages in case of a one-step mode. It may need to operate as a two-step clock. In this case, as an example, it must modify the passing `Sync` messages as two-step and send its own `Follow_Up` message. [19]

TCs never issue their own `Sync` messages as masters because their ports never enter any of the introduced port states. Two types of TCs are defined in the 2019 revision of the protocol: [12]

E2E TC is a variant which allows all the delay request-response mechanism messages through. For one-step masters, the `correctionField` of `Sync` messages are adjusted; for two-step masters, only the `Follow_Up` messages are adjusted. Note that the 2019 revision of the protocol explicitly states that the messages are retransmitted as a new packet and are not just adjusted;

P2P TC variant drops all the delay request-response mechanism messages. It measures the ingress `meanLinkDelay` of all its ports and adjusts the passing synchronization messages with it in addition to the `residenceTime`. [12]

Figure 3.19 illustrates two TC hierarchies because these types cannot coexist in one [12].



■ **Figure 3.19** PTP end-to-end versus peer-to-peer transparent clock. Depicted is the difference where in E2E mode, the TCs are fully transparent to the traffic, whereas in the P2P mode, they must actively participate in the P2P delay measurement. [12]

3.7.12 Best Master Clock Algorithm

The Best Master Clock Algorithm (BMCA) of PTP builds the master-slave hierarchy in the network. *Default* BMCA is provided, though profiles are allowed to implement *alternate* BMCA. The same algorithm continuously runs on all the domain OCs and BCs, ensuring that the best clock will become the grandmaster and that the domain will react to clock quality and network topology changes. [12] The BMCA consists of two algorithms: data set comparison algorithm (DSC) and state decision algorithm (SD) [19].

All clocks provide quality attributes by periodically issuing the `Announce` general message [12]. Each pass of this message through BC increments the included counter field `stepsRemoved`, initially set to 0. The DSC pairwise compares the received attributes of all clocks and the clock itself and chooses the best one. This is done per port. Then, the SD chooses the next state of the port. If a better clock is discovered, the port enters the slave state. If the port is the best one, it enters the master state. [19]

Two ports may determine the same remote clock as being the best one. The port which received the message with lower `stepsRemoved` counter is selected as *topologically* better and enters the slave state. The other port enters the passive state. This unambiguously selects one port when cyclic paths in the network are not removed by a protocol other than PTP. An example illustration is provided in Figure 3.20. [19]

The data set comparison algorithm does a pairwise comparison in this order:

priority1 user-configurable priority;

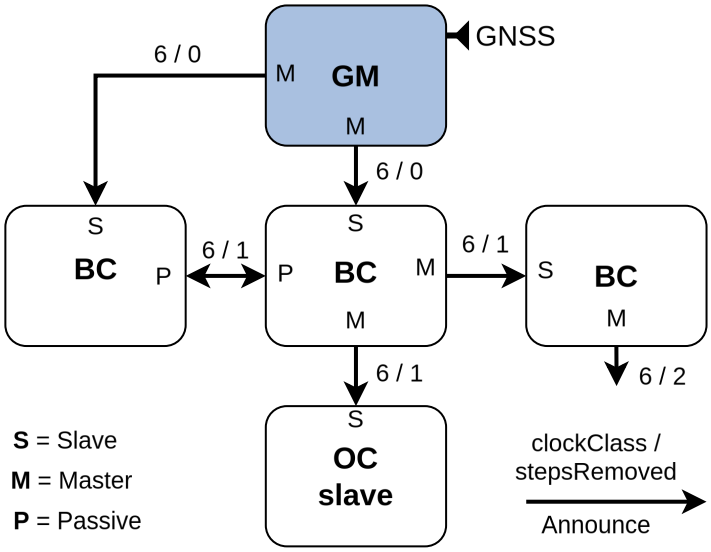
clockClass attribute defining TAI traceability;

clockAccuracy attribute defining the accuracy of the LPC;

offsetScaledLogVariance attribute defining the stability of the LPC;

priority2 finer-grained user-configurable priority;

clockIdentity field, which is used as a tie-breaker. [25]



■ **Figure 3.20** PTP BMCA algorithm. Depicted is the result of the BMCA algorithm simplified to the `clockClass` and `stepsRemoved` fields. [12]

3.7.13 IEEE 802.1AS profile

The generalized PTP is a profile defined for use in TSN. It is a strict subset of the complete IEEE 1588. The gPTP standard document defines the key differences between IEEE 802.1AS and IEEE 1588: [26]

- The only supported transport mechanism is Ethernet MAC PDUs utilizing Ethernet addressing. No other transports are supported. [26]
- gPTP specifies a media-independent sublayer, which simplifies the integration of technologies with different media access protocols into one timing domain, such

as IEEE 802.11. Each medium has its media-dependent layer specified. In IEEE 1588, this is optional. [26]

- Two types of domain nodes are specified instead. *End instances* correspond to OCs and *relay instances* correspond to TCs. However, relay nodes can send `Sync` messages and invoke the BMCA to let their ports enter any of the states. [26]
- Communication is always done only between gPTP instances. Non-relay instances cannot be used to pass gPTP traffic. Thus, non-PTP devices are also not allowed anywhere between gPTP nodes. [26]
- Only the peer-to-peer delay mechanism is allowed. [26]
- gPTP requires two-step processing while declaring one-step processing as optional. IEEE 1588 allows both of them to be required. [26]
- All instances are required to be logically syntonized. This may be done by the one-way PTP mechanism. [26]

The standard requires all participating time-aware systems separated by six or fewer nodes, seven hops at maximum, to be peak-to-peak synchronized within 1 μs. The rate of `Announce` and `Pdelay_Req` messages is once per second. `Sync` messages are issued eight times per second. [12]

3.8 White Rabbit

White Rabbit (WR) is an exemplary state-of-the-art application of the presented synchronization technologies. The White Rabbit Project is a: "multilaboratory, multicompany and multinational collaboration to develop new technology that provides a versatile solution for control and data acquisition systems" [27] started by the Conseil européen pour la Recherche nucléaire (CERN) of Meyrin, Switzerland, the European Organization for Nuclear Research in English. [27]

It is an open-source project defining the *White Rabbit Network*. The network is an Ethernet-based bridged LAN with VLANs, which synchronizes frequency using SyncE and phase and ToD using an extension of PTP. It features picosecond synchronization accuracy between thousands of nodes with an average distance of 10 kilometres over gigabit Ethernet links. The extension was merged into the 2019 revision of PTP. [27]

WR employs numerous asymmetry corrections. For example, the asymmetry introduced by the layout of the circuit board of the network element is assumed to be static and accordingly compensated for by measuring its amount. Medium asymmetry of fiber runs between nodes is measured. Changes in propagation speeds through fiber runs based on cyclical changes, such as temperature changes during the day, are corrected. The last example is calibrating the physical links utilizing specialized network equipment. All these and other corrections help WR achieve such a level of accuracy. [12]

Chapter 4

PikeOS driver integration

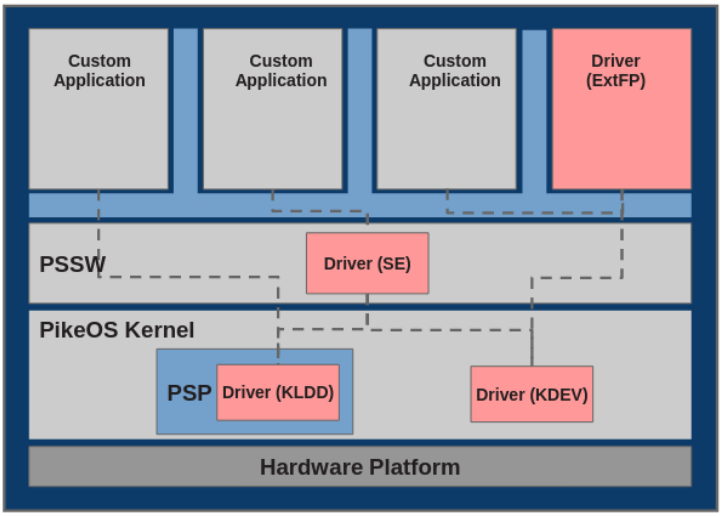
This chapter introduces the hardware used, specifically the TQ-Systems GmbH MBa8MPxL single-board computer and the ENET_QOS Ethernet controller. The primary topic is introducing the PikeOS Driver Development Kit framework and analysing the controller's features selected for integration.

4.1 PikeOS drivers

Drivers in PikeOS are used to extend the system I/O functionality. That provides additional functionality to operations like `read` and `write`. The language of choice is the C language. Drivers implemented in the kernel, Kernel Drivers (KDEV), are gate providers that can be accessed using files or other APIs. Drivers accessible via the file system are called file providers or volume providers. File providers may be implemented in the PSSW, in which case they are called Internal File Providers. These are called External File Providers (ExtFP) inside the user partition. Since PikeOS 4.0, the recommendation is to choose between Kernel Drivers and External File Providers. This hierarchy is depicted in detail in Figure 4.1. [6]

4.1.1 Configuration

The PSSW and user-level drivers may be configured using the PikeOS property file system (PropFS). It is implemented as an internal file provider in the PSSW and is intended to provide read-only configuration data. Access is done through system calls using predefined paths starting with `prop:` prefix to select this filesystem. Configuration is provided through strongly typed binary properties. Possible properties for a given driver are defined using XML files, providing the structure and types of all needed properties and default values of optional properties. All of these XML files are then merged into a property tree and embedded in the target's ROM image. [6]



■ **Figure 4.1** PikeOS Driver Placement. The hierarchy of different PikeOS driver types is depicted. The user-partitioned External File Provider is in the top right corner. The Kernel Drivers are on the bottom. [6]

Kernel Drivers are configured using a binary compiled from another specialized XML file providing set values for all required properties. This form of configuration has an advantage for array-like properties. Access to these is much faster using the precompiled binary because, in the case of PropFS, each member access requires its system call. The disadvantage of this decision is that there are less flexible options for changing the underlying format. PropFS may be changed or extended while providing the same functionality without any API change. On the other hand, the binary format must have its version explicitly announced and handled. Example snippet of a driver configuration is provided in Listing 4.1. [6]

```
<ParameterTable>
  <Separator display="Driver Specific Configuration"/>
  <Parameter name="BCAST_ENABLE" type="boolean" value="true"/>
</ParameterTable>
<Romimage>
  <properties>
    <prop_dir name="config/provider/$(DEP:PROVIDER:PREFIX)">
      <prop_dir name="priv/io/$(IO_ID)">
        <prop_bool data="$(BCAST_ENABLE)" name="bcast_enable"/>
      </prop_dir>
    </prop_dir>
  </properties>
</Romimage>
```

■ **Code listing 4.1** Snippet of PikeOS driver property XML specification. Depicted is the definition of broadcast reception enable boolean property for the ENET_QOS driver.

4.1.2 Separation

All `open`, `close`, and `stat` requests from user space are handled by an inter-process communication (IPC) call through the PSSW. This decision enables access control and makes access time the same for all drivers. Exempt from this are `open` and `close` calls for volume providers. For them, only the `mount` operation is tunnelled through the PSSW for access control. All other operations like `read`, `write`, and `ioctl` are handled using IPC requests by the drivers. These involve an address space switch to the driver's address space. Kernel Drivers are exempt from all PSSW tunnelling because they use system calls directly. These are faster because no address space switch is required. [6]

Because ExtFP and volume providers reside in their user partition and their own address space, any failure during their execution does not propagate to other partitions. This makes them non-critical to the system. That is other than critical inter-partition dependencies. Internal file providers run in the address space of the PSSW and are thus on the same criticality level as the PSSW. That is the highest in the system. None of these have access to the kernel memory space. Kernel Drivers are also critical because their failure will affect the whole system. In addition, they are also given access to the kernel memory space. [6]

4.1.3 Resource access

User-space drivers have routed interrupts by blocking. This means that interrupt handling may involve address space switches. The implication is that hardware that needs an immediate reset of a given interrupt cannot be driven from user space. In contrast, Kernel Drivers can be used for such hardware. Specific callbacks for interrupts may be installed. These callbacks operate in a limited environment where only wake-up of other threads and data transfers are allowed. [6]

Access to the I/O memory region or other resources may be granted to user space drivers at their request. Exempt from these are resources running in supervisor mode, like high-resolution timers. Kernel drivers are given access to all resources. [6]

All user space drivers may use all the PikeOS threading, synchronization and blocking functionalities. Worker threads in the driver partition may be created to handle background tasks. Internal file providers are exempt from this because they run in the PSSW partition daemon thread, which is not blockable. Their worker threads can block if the communication with the partition daemon is not blocked. This means that, for example, mutexes cannot be used because they block. Kernel Drivers can block because they run exclusively when invoked through a system call. On the other hand, the kernel API provides only a limited set of blocking calls. Kernel Drivers are not allowed to start any worker threads. [6]

4.2 Driver Development Kit framework

The DDK stands for Driver Development Kit. The runtime environment in which drivers execute is referred to as the driver framework. [28] This framework consists of two categories of elements:

Driver Entry Points represent the public interface of the driver in the direction of the rest of the system. The framework defines the semantics of these;

Driver Services are functionalities provided by the driver framework which the drivers themselves can use. [28]

No specific behaviour is defined for any driver by the framework. So-called device classes do this. These classes group drivers that handle I/O devices of similar properties. [28] Seven specific classes are defined:

CHAR class for generic drivers;

SER class for serial universal asynchronous receiver-transmitter (UART) drivers;

NET class for Ethernet drivers;

DIO class for general purpose input output (GPIO) drivers;

CAN class for CAN bus drivers;

BLK class for storage drivers;

WDT class for watchdog timer drivers. [28]

The framework furthermore differentiates two types of devices:

I/O device is a hardware device independent of its use of registers and interrupts from all other devices. The driver is assigned this device to manage;

Logical device is an object which is provided to the user clients. There is no strict requirement for 1:1 mapping between the managed I/O and logical devices. Driver design may allow 1:N mapping. [28]

4.2.1 DDK Network Class

The framework's network class is used for drivers which manage Ethernet controllers. It defines data types and framework services that ease the implementation of such drivers. Relevant services are introduced in the implementation chapter. [28]

The file system API which any network driver must provide is:

vm_open() to select operation mode and start the controller;

vm_close() to stop the controller;

vm_read() to receive an Ethernet frame;

vm_write() to transmit an Ethernet frame;

vm_ioctl() to set runtime configuration and receive status. [28]

This class introduces the concept of virtual Ethernet devices. Applications see these devices as Ethernet ports with their own MAC address, other than the controller's physical port. This enables the use of one Ethernet device by multiple applications. That is because opening the device implies exclusive access to it. Two types of mutually exclusive device operation modes are defined. [28] Selection is made by opening a device of a given type:

Real device mode is selected when the real device provided by the driver is opened. The partition has exclusive access to the controller's physical port. The broadcast domain in this mode is strictly the physical network. This is an instance of 1:1 I/O device mapping;

Virtual mode is selected by opening the virtual Ethernet device. In this mode, all opened virtual devices share the physical Ethernet port. Broadcast domain, in this case, extends the physical network with all the virtual devices. This is an instance of 1:N I/O device mapping. [28]

The `ioctl` client API provides the state of the device statistic counters and link status. It allows querying and setting the hardware address used. Finally, the runtime configuration of multicast reception is done through this interface. [28]

4.2.1.1 Address filtering

Each network driver decides in its implementation if perfect software filtering or imperfect hash filtering is done for frame reception based on the target MAC address. Typically, hardware such as the ENET_QOS controller implements only imperfect hash filtering if multiple MAC addresses are requested. That is the case when multiple virtual devices are used. This means further processing is needed if only matching packets should be seen by the driver clients. This decision makes driver development more flexible. Either the driver's higher performance may be implemented using imperfect hardware filtering, or higher security is chosen to ensure the clients see only frames that belong to them. [28]

4.2.1.2 Read and write frame transfers

Transfer of Ethernet frames using the `read` and `write` calls is done one Ethernet frame at a time. It is not possible to transfer multiple frames at once. The configured maximum transfer unit (MTU) determines the maximum size of such a frame. The recommended value for drivers without jumbo frame support is 1522. If the buffer provided to the read operation is smaller than this value, the driver can fill the buffer and discard the rest of the frame. These operations can be asynchronous with the wire frame reception and transmission. That is, the driver is allowed to buffer the frames internally. If not enough space is left to hold the frames, any following frames can be dropped and counted. [28]

The driver operations may be implemented as blocking. This means waiting up to a configured period for a frame to become available for a `read` operation. This means waiting for enough space for the transmission for a `write` operation. In virtual operating mode, this is true only for frames going to the physical network. Frames going to other virtual devices never fail to transmit. Nevertheless, insufficient resources may lead to a drop of these frames on the receiver side. [28]

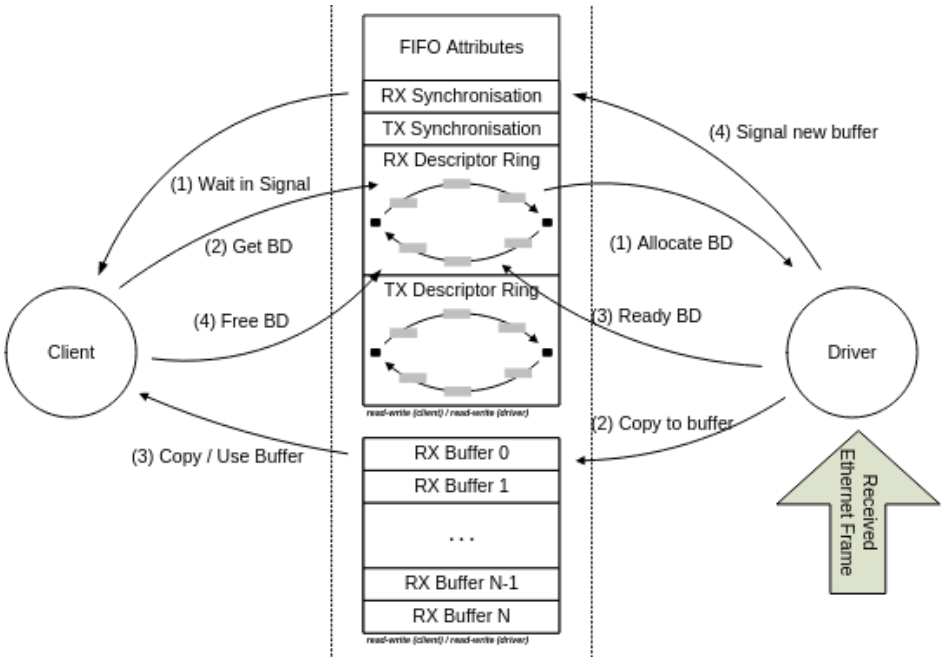
In nonblocking mode, the read operation returns immediately with a timeout error if no frame is available. The write operation does the same if insufficient space for the frame transmission is available. [28]

4.2.1.3 Shared buffer frame transfers

Frame transfers between the driver and client may be done using shared buffer communication (SBUF). This mode establishes a shared memory region, which the driver provides, and the application maps. This API should offer higher throughput due to the minimized context switching. The application uses an `ioctl` call in runtime to switch to this mode. [28]

Two FIFO rings are established for packet reception and transmission. The client is responsible for managing the TX ring. The driver is responsible for the RX ring. In contrast to the read and write mode, both sides can operate on multiple frames simultaneously. These rings hold buffer descriptors, which hold an index into the shared memory region where the buffers reside. The size of the buffers is set to the configured MTU. The size of these rings is user-configurable, with a maximum size of 511 descriptors. [28]

When the driver receives a frame, a vacant buffer in the RX ring is allocated. This removes its descriptor from the ring, and the driver copies the frame data into this buffer. Then, the driver puts its descriptor back into the ring and signals to the client that a new packet is ready. The client, woken up by this, deques the new descriptor and may immediately use the frame data in the buffer. After this, the client puts the descriptor back into the ring. The same holds for frame transmission, but only the driver and client switch roles. Note that both parties modify both rings and could thus corrupt the ring of the other party. The case of frame reception is depicted in Figure 4.2. [28]



■ **Figure 4.2** PikeOS shared buffer communication frame reception. The driver allocates a buffer, and data is copied into it. The filled descriptor is returned to the ring, and the client is notified. The client removes the ring upon notification, uses the data and returns the buffer to the ring. [28]

4.2.1.4 Configuration properties

Configuration of network class devices is done using properties in PropFS. The Ethernet I/O device must have configurable link settings. These are listed in Table 4.1. Two types of configurations are defined for real and virtual devices. The configurable parameters for real and virtual devices are listed in Table 4.2. The driver may also expose any specific configuration properties. An example of such property for enabling broadcast frame reception in the ENET_QOS driver was shown in Listing 4.1. [28]

■ **Table 4.1** PikeOS Ethernet I/O device configuration properties. [28]

Pathname	Type	Decription	Default
`aneg`	bool	Autonegotiation enabled or not.	True
`duplex`	bool	Full-duplex enabled or not.	True
`speed`	uint32	Link speed in Mbps.	100

4.2.2 DDK Network High Level Module

The DDK Network High Level Module implements the hardware-independent part of a DDK Ethernet driver. It can be used for ExtFP drivers. It is provided as a binary

object linked to the hardware-specific driver during build. It implements all the features introduced in the previous section. These features include configurable software RX FIFO, handling of blocking and nonblocking modes, real and virtual device access and the SBUF mode of operation. [28]

The complete interface, which defines the data and control flows between the low-level and high-level modules, is depicted in Listing 4.2. It is analysed in detail in the implementation chapter. [28]

4.2.2.1 Hardware address generation

When operating in the virtual device mode, each interface needs to have its hardware address assigned to receive and transmit frames. These addresses can be configured by the user or autogenerated based on the real device address. This is done in a way to prevent address collisions in the network. [28]

MAC addresses are constructed from two parts: the organizationally unique identifier (OUI) in the three most significant bytes and the network interface controller (NIC) in the three least significant bytes. The autogenerated address keeps the NIC part. The most significant byte is set as the interface number shifted left by two bits. This byte's second least significant bit is set to mark the address as locally administered. The following two bytes are set to ASCII string `p4`. An example of an address generated by this algorithm is `02:70:34:aa:bb:cc`. [28]

```
/* Provider entry points. */
drv_char_init_prov_t      init_prov;
drv_char_init_io_dev_t    init_io_dev;
drv_char_init_logic_dev_t init_logic_dev;
drv_char_init_complete_t  init_complete;

/* Logical device entry points. */
drv_hlnet_start_dev_t start;
drv_hlnet_stop_dev_t  stop;
drv_hlnet_transmit_t  transmit;
drv_hlnet_get_mac_t   get_mac;
drv_hlnet_set_mac_t   set_mac;
drv_hlnet_set_mcast_t set_mcast;
drv_hlnet_ioctl_t     ioctl;
```

■ **Code listing 4.2** API between DDK high-level and low-level modules. Listed are all callbacks the low-level module must implement. [28]

4.2.2.2 Multicast management

The high-level module abstracts the network class multicast API from the low-level device. The filtering is implemented using the imperfect option for performance reasons.

Table 4.2 PikeOS network device configuration properties. At the top are properties common to both device types. Following are additional real device properties. Last are virtual device properties in addition to the common ones. [28]

Pathname	Type	Decription	Default
		Real/Virtual	
`mac_address`	mac	MAC address.	00:00:00:00:00:00
`receive_queue_depth`	uint32	Software RX FIFO size.	32
`send_queue_depth`	uint32	Software TX FIFO size.	0
`enable_mcast`	bool	Enable multicast RX/TX.	False
`mcast_table_size`	uint32	Size of multicast MAC table.	No default.
`mbuf_pool_size`	uint32	Software buffers available.	No default.
		Real	
`vlan_dev`	bool	Enable VLAN routing.	False
`vlan_tag_ethertype`	uint32	0x8100 or 0x88a8.	No default.
		Virtual	
`vlan_id`	uint32	802.1Q tag	No default.
`vlan_rx_untag`	bool	Strip tag on RX.	True
`vlan_tx_tag`	bool	Add tag on TX.	True
`vlan_prio`	uint32	802.1Q priority value.	0.

That is because standard hardware also implements this option. This means that a hash table created from the configured addresses is used. Clients may thus receive frames that do not belong to them. Clients must account for this, but the low lever driver can still try to minimize the load on the software filtering by using the hardware perfect filtering if such capability is available. [28]

Addresses can be individually added, deleted or the table reset. The size of the software table is limited by the `mcast_table_size` property. Multicast handling must be enabled on the real device and all virtual devices, which should accept multicast traffic using the `enable_mcast` property. The table is created in the virtual device mode by combining tables from all active virtual devices. [28]

4.2.2.3 VLAN based routing

The high-level module also implements VLAN-aware mode when operating in the virtual device mode. In this mode, not only the MAC address of each virtual device is taken into account during software frame filtering, but also the 12-bit VLAN identifier. The tagging scheme is configurable. This means that not only 802.1Q is supported, but also 802.1ad. Frames without any tag are processed as if having tag 0. [28]

During reception, frames are delivered to the device with a matching tag and MAC address. If no such device exists, the frame is dropped. Broadcast frames are delivered

to all devices with matching tags. Multicast frames are delivered to all devices that are registered for the address and have a matching tag. The tag may be stripped. [28]

During transmission, frames are delivered to the virtual device which matches the destination MAC and tag. Otherwise, the frame is sent to the physical network. Broadcast frames are forwarded to all other virtual devices with matching tags and the physical network. Multicast frames are forwarded to all devices that are registered for the address and have a matching tag. The tag may be optionally added before transmission. [28]

4.3 TQ-Systems MBa8MPxL SBC

The integration work is performed on the MBa8MPxL revision 01XX single-board computer (SBC) developed by TQ-Systems GmbH based on their TQMa8MPxL platform. This board is shown in Figure 4.3. [29] It is based on the i.MX 8M Plus platform developed by NXP. This platform is designed as a starting base for integrators focusing on advanced multimedia, machine learning, and industrial automation applications with high reliability in Industry 4.0. [30]

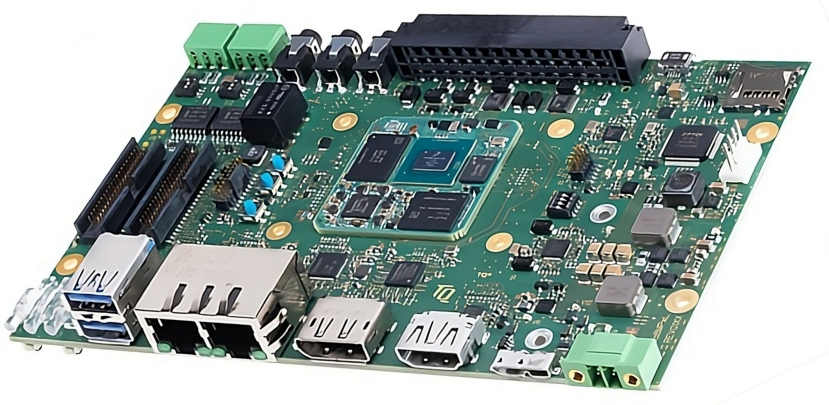


Figure 4.3 TQ-Systems GmbH MBa8MPxL SBC. The RJ-45 connector on the right is connected to the ENET_QOS controller. The USB 2.0 Micro-B connector next to the SD card slot in the top right corner provides access to the board's serial console. [31]

The MBa8MPxL board features four ARM Cortex-A53 cores with an industrial temperature range from -40 °C to 105 °C and 2 GiB of LPDDR4 memory. The board's I/O features a wide range of audio and video connectors, which are not crucial for the focus of this thesis. The essential components are the ENET_QOS ethernet controller and the Texas Instruments DP83867IS physical layer chip. The controller supports advanced features, including quality of service, audio video bridging and the IEEE 1588 PTP protocol used for time synchronization in TSN. [29]

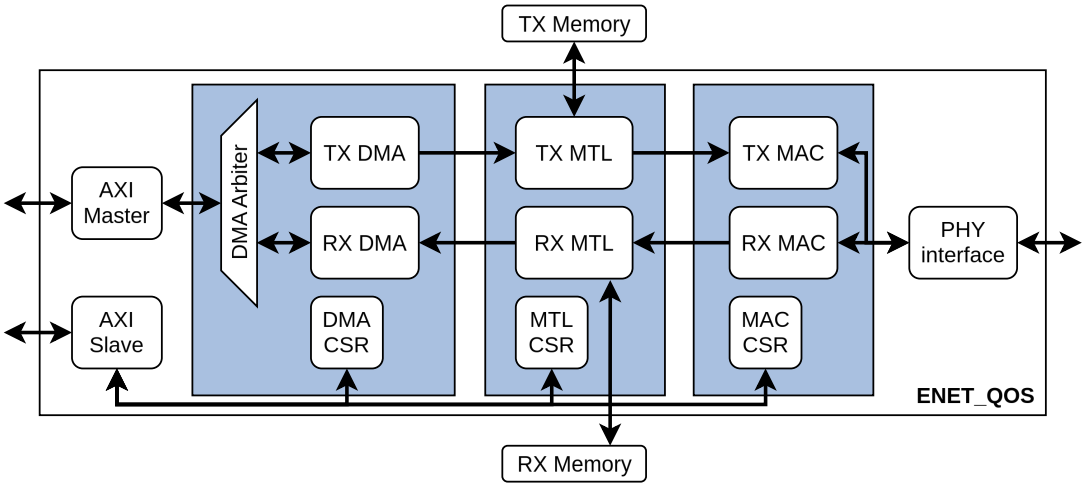
PikeOS already supports this target in its board support package (BSP) for this target. The BSP initializes all peripherals using the Das U-Boot bootloader and supports selected features using PikeOS drivers.

4.4 ENET_QOS controller

The ENET_QOS module is a fully compliant IEEE 802.3-2015 ethernet controller. In addition to the default features of the IEEE specification, it supports a range of additional ones. A subset of these features applicable to the PikeOS integration and the IEEE 1588 PTP protocol is introduced starting with Section 4.4.2. [32]

4.4.1 Architecture

The controller is split into three blocks: direct memory access, MAC transaction layer and the MAC. These provide the core interfaces, functionalities and protocol support. Each block has its control and status register (CSR) set used for configuration and querying the block's state. Figure 4.4 illustrates the controller's block diagram. [32]



■ **Figure 4.4** ENET_QOS Ethernet controller block diagram. Depicted are the three core blocks of the controller and how data flows between them. The registers are accessible separately from the data path. Notable are the frame queue memories, which are not part of the controller. [32]

4.4.1.1 Direct memory access controller

Direct memory access (DMA) access is done using either the Advanced Extensible Interface (AXI) or the AMBA High-performance Bus (AHB). These busses are defined in the ARM Advanced Microcontroller Bus Architecture (AMBA) [32]. The AXI bus is the one used on the MBa8MPxL [33].

The DMA controller has independent transmit and receive engines. The engines move the Ethernet frame data between the system memory and the MAC transaction layer (MTL). A descriptor ring of configurable length from 4 to 1024 is used to minimize the application intervention. The DMA controller supports up to 5 TX and 5 RX independent descriptor rings, also called channels. These may be used to split different

traffic priorities based on the 802.1Q VLAN tags. The driver implementation for basic functionalities and PTP does not need to handle traffic priorities and is limited to only the DMA queue 0. [32]

Each descriptor may point to up to 2 data buffers in physical memory. These buffers do not have any restrictions on their start address alignment, but operations will be done using the DMA bus alignment. Data, therefore, start at the correct address, but dummy bytes are still read from or written at the address aligned. The PikeOS high-level module abstraction layer provides buffers aligned to the DMA bus width. The structure of the descriptor ring and the descriptors themselves are introduced in the implementation chapter. [32]

4.4.1.2 MAC transaction layer

The MTL is a simple FIFO memory between the DMA and MAC in the controller. It regulates the flow of packets between these two blocks. Like the DMA, it features two data paths, one for TX and one for RX. Each path has eight kibibytes (KiB) of memory on the MBa8MPxL SBC. [32] Both of the data paths can operate in two modes:

Threshold mode forwards data from and to the MAC once the configured number of bytes is stored in the queue. Thus, frames may span multiple DMA descriptors;

Store-and-forward mode waits for the complete packet to cross the queue boundary or for the queue to fill up. Only valid packets are forwarded to the MAC on the RX data path. Thus, one frame equals one DMA descriptor. [32]

The driver implementation uses the store-and-forward mode. This mode makes the descriptor ring management more predictable when introducing PTP traffic. However, this decision limits the driver's maximum supported frame size to the size of the FIFO.

4.4.1.3 MAC

The controller's MAC transfers complete frames between the PHY and the controller. After stripping the physical layer preamble, the MAC applies configured filtering and other actions for supported protocols. Selected features of the MAC are introduced in the following sections. [32]

4.4.1.4 Interrupts

For the reduction of the CPU load, the controller supports interrupt coalescing. Each of the three introduced blocks generates interrupts configured using their enable registers. Frame reception and transmission are captured per DMA queue. Interrupts are not queued in the controller and the respective blocks. Because of this, the driver must handle the event firing again before even the first occurrence is handled. [32]

4.4.2 PHY support

The media independent interface (MII) is a simple interconnection between MACs and PHYs. It is widely known as the IEEE Clause 22. [34]

4.4.2.1 MII data exchange

The MII interface provides 10 Mb/s and 100 Mb/s full duplex operation. The transfer is done across four-bit wide independent receive and transmit paths. Each path also includes its own delimiter, error and clock signals. The PHY drives the transmit clock based on the physical layer link speed, 2.5 MHz for 10 Mb/s and 25 MHz for 100 Mb/s. The receive path extends this with collision detection and carrier sense signals for carrier sense multiple access (CSMA) on the physical layer. The receive clock is recovered from the incoming signal using a PLL in the PHY. That is 16 signals in total. [34]

Because the 16 MII signals are separate for every PHY connected to the MAC, high port density systems require a lower pin count design. Furthermore, the system clocking design is complex because of the separate TX and RX source synchronous clocks. The reduced media independent interface (RMII) implements this simplification. The clocking is done using a system synchronous 50 MHz clock for both TX and RX shared across all ports. The data paths are reduced to two bits. That is why the clock frequency is doubled from the 25 MHz used in 100 Mb/s operation in MII. The 10 Mb/s operation is done using oversampling. That is repeating the same signal for ten clock cycles. To further reduce the pin count, the collision detection signal is removed. The carrier sense signal is multiplexed with the RX delimiter on alternate clock cycles as a last reduction. Consequently, RMII requires seven signals, and the clock is shared across all ports. [35]

To support 1000 Mb/s operation between the PHY and MAC, a gigabit media independent interface (GMII) may be used. Its design is the same as that of MII, only extending the data paths to eight-bit wide and enabling a 125 MHz frequency for the TX clock operating in 1000 Mb/s mode. This clock has a signal separate from the MII TX clock signal. It is used only in the 1000 Mb/s mode to be backwards compatible with MII. That is 25 signals in total. [34]

For the same reasons as with MII, reduced gigabit media independent interface (RGMII) reduces the pin count of GMII. The data paths are reduced to four bits wide, and double data rate (DDR) signalling is used instead. That is sampling the data on both the rising and falling edge of the clock signal. Source synchronous clocking is still used, but the extra TX gigabit clock signal is removed. The CSMA signals are removed. The TX and RX error and delimiter signals are multiplexed using both clock edges. That is 12 signals in total. [35]

4.4.2.2 MII management

The control and status information exchange between the PHYs and MACs is done using the management data input/output (MDIO) bus. This bus is shared between all the PHYs and MACs. Two signals drive it: the MDIO signal and the management data clock (MDC) signal. The clock is not free-running and has an upper limit of 2.5 MHz. Each PHY on this bus has a five-bit wide address assigned. In its design, it is very similar to the Inter-Integrated Circuit (I2C) bus. [36]

All PHYs shall support the basic register set consisting of two registers, the Control Register 0 and Status Register 1. Gigabit PHYs shall also incorporate the Extended Status Register 15. Registers 2 through 14 are part of the extended register set defined for the auto-negotiation protocol. That is IEEE protocol Clause 28 or Clause 37. The whole register set is listed in Table 4.3. The specific contents of these registers needed for the controller driver are introduced in the implementation. [34]

Table 4.3 Basic and extended MII register set. [34]

Address	**Name**	**Basic/Extended**	
		MII	GMII
0	Control	B	B
1	Status	B	B
2,3	PHY Identifier	E	E
4	Auto-Negotiation Advertisement	E	E
5	Auto-Negotiation Link Partner Base Page Ability	E	E
6	Auto-Negotiation Expansion	E	E
7	Auto-Negotiation Next Page Transmit	E	E
8	Auto-Negotiation Link Partner Received Next Page	E	E
9	MASTER-SLAVE Control Register	E	E
10	MASTER-SLAVE Status Register	E	E
11	PSE Control register	E	E
12	PSE Status register	E	E
13	MMD Access Control Register	E	E
14	MMD Access Address Data Register	E	E
15	Extended Status	Reserved	B
16-31	Vendor Specific	E	E

Clause 22 management data is exchanged using frames in two types of operations. That is operation read `10` and operation write `01`. Communication starts from the MAC to the PHY with a preamble of 32 bits of `1`, followed by the start field '01', the operation code, the five-bit wide PHY address and the five-bit wide register address. In the case of a write, the MAC follows with bits `01` and sends 16 bits of data. That is the MII register width. In the case of a read, it releases the MDIO line and waits on it for the data from the PHY. The frame format is shown in Figure 4.5. [34]

Frame fields							
	PRE	ST	OP	PHYAD	REGAD	TA	DATA
READ	1…1	01	10	AAAAA	RRRRR	Z0	DDDDDDDDDDDDDDDD
WRITE	1…1	01	01	AAAAA	RRRRR	10	DDDDDDDDDDDDDDDD

Figure 4.5 MII management frame format. [34]

4.4.2.3 Controller MII support

The RMII and RGMII interfaces are both supported for the interconnection of the MAC layer in the controller and the PHY chip. The interface selection cannot be changed through the controller's registers. It is done using a physical selection pin sampled during the controller reset. RGMII is selected on the MBa8MPxL. [32]

Because PikeOS itself does not have an abstraction layer over the MDIO bus or PHY handling, this must be implemented in the controller's driver using the provided registers. Clause 22 management is provided through the MDIO address register at offset `0x200`, setting up the frame structure. The MDIO data register at offset `0x204` holds the data from the PHY after the read operation or the data to submit in case of a write operation. [32]

4.4.3 IEEE 802.1Q support

The controller supports the IEEE 802.1Q VLAN tagging standard. Both single and double VLAN tagging are supported. The inner tag, known as customer VLAN (C-VLAN), and the outer tag, known as service VLAN (S-VLAN), may be replaced or removed before transferring the frame to the application. On the transmit side, both tags can be inserted by the controller. It is important to note that the controller requires the ethernet header type to be set to 0x8100 or 0x88a8 when VLAN tagging is used. The S-VLAN ethernet type is thus not configurable. This follows the IEEE specification but makes the controller less versatile. If only one VLAN tag is used, it is considered an outer tag. The structure and difference of such frames are shown in Figure 4.6. [32]

The VLAN tagging support is also important because it changes the behaviour of the controller's maximum transfer size (MTS) set. That is the maximum ethernet frame size in bytes, including the frame check sequence (FCS), which the controller can transfer. If VLAN tagging is used, the controller considers this limit to be increased by 4 or 8 bytes per frame. That is the size of one or two 802.1Q tags, based on the tagging mode. [32]

4.4.4 Filtering features

The controller supports a four-layer filtering sequence of incoming frames. Any frame that fails the given filter is immediately discarded. In the opposite case, it is passed to

the upper layer filter or the application in the case of the last filter. The promiscuous mode may be enabled to skip all the filters and enable the system supervision. [32] The filtering layers are:

Ethernet filter is the first to receive the frame. It filters frames based on the source or destination MAC address. Broadcast frames may get dropped or passed. Unicast frames may get filtered using the perfect filter, in which case all the 48 bits of the addresses are checked. Any additional unicast addresses may get filtered using imperfect hash filtering. A hash table of size 64 is used in this case. Its bucket is considered a pass if it is set to 1. The indexing into this table is done using the six most significant bits (MSB) of the IEEE 802.3 FCS algorithm applied to the address. The same applies to multicast frames, which share the hash table with the unicast. However, unicast and multicast filtering are done and enabled separately. Group filtering can be done by comparing only specific bytes of the perfect MAC setup;

VLAN filter is queried after the Ethernet one. It enables the same filtering as can be done on the Ethernet layer. Eight VLANs may be filtered perfectly. Hash filtering can be done using four MSBs of the tag's FCS in a 16-bucket hash table. Only the 12-bit identifier of the tag can be compared instead of the complete 16 bits;

IP filter filters the source or destination IP address of up to eight perfect matches. Both IPv4 and IPv6 addresses are supported;

TCP/UDP filters the Transmission Control Protocol's (TCP) and UDP's source and destination ports. The configuration is shared with the IP filter for eight perfect matches, but the TCP/UDP filtering can be selectively enabled per slot. [32]

All the filtering options have their inverse enable flag, which inverts the outcome of the filter. For example, changing the hash table filtering from pass mode to drop mode. That is drop when the frame's index bucket is set to 1. [32]

SMAC	ETYPE/SIZE	PAYLOAD	FCS

SMAC	802.1Q (0x8100)	ETYPE/SIZE	PAYLOAD	FCS

SMAC	802.1ad (0x88a8)	802.1Q (0x8100)	ETYPE/SIZE	PAYLOAD	FCS

■ **Figure 4.6** 802.1Q tagged ethernet frame. The preamble, destination address, and interframe gap are removed for conciseness. From top to bottom, there is a frame without a tag, an 802.1Q single-tagged frame in the middle, and an 802.1ad double-tagged frame at the bottom. [37]

4.4.5 PTP features

The MAC block supports all the introduced clock types of the 2008 revision of PTP in both the one-step and the two-step operation modes. The one-step transmission is selected per frame in the frame's DMA descriptor. [32]

4.4.5.1 Local PTP clock

The MAC features one instance of a local PTP clock used for timestamping and provided to the PTP software in one of two possible configurations chosen by the controller's hardware implementor:

External time source connected through a 64-bit wide interface. This configuration allows the MAC to use a possibly higher-quality external time source while limiting the PTP seconds to 32 bits instead of the standard 48 bits;

Internal time source possibly providing the full 80-bit PTP timestamps. The upper 16 bits of seconds may still get disabled. In this mode, a clock signal must drive the internal time source. This option is used on the MBa8MPxL driven by a 100 MHz, ten nanosecond period clock signal. [32]

The internal time source further allows two modes of operation precision:

Digital rollover mode limits the 32-bit nanosecond field to a maximum value of $10^9 - 1$. After this value is reached, it rolls over to 0. This provides a 1 ns resolution;

Binary rollover mode limits this field to a maximum value of `0x7FFFFFFF` after which it rolls over to 0. This provides ∼0.465 ns resolution. [32]

The software may update the internal time source by setting the system time or providing an offset added to the system time. This is referred to as the coarse method, as it may introduce extreme changes or jitter to the system time. The other option is changing the flow of time of the internal clock through a so-called `addend` register. This is referred to as the fine method. The decision about which method should be used depends on the capabilities of the PTP software used. The driver implementation on MBa8MPxL uses the fine method, which is examined in more detail in the implementation chapter. [32]

4.4.5.2 Timestamping support

The timestamp on frame reception is captured per standard at the leading edge of the first bit of the first octet following the frame's SOF. This may be selectively done for all frames, only PTP frames, selected PTP frames or only PTP event frames. The captured timestamp is returned in the descriptor following the last one for the received

frame. Thus, each timestamped frame spans exactly two descriptors in store-and-forward mode. This simplifies mapping the captured timestamps to their respective frames. [32]

The same timestamping rules hold for frame transmission, with the only difference being that the timestamp is captured after SOF is transferred and not received. Timestamping for transmitted frames is selected per frame in the frame's DMA descriptor. The captured timestamp is returned in the same descriptor. [32]

Because the timestamping in both directions is far from the network boundary, it must be corrected. Known PHY latency may be provided to the controller so that it can be corrected automatically. The controller provides internal ingress and egress delays, which must be read on link speed change and written into a second set of registers. Lastly, the errors introduced by clock domain crossing (CDC), the difference between the PTP clock signal and the selected MII interface, are specified as twice the period of the PTP clock. [32]

4.4.5.3 PikeOS API

The introduced stable PikeOS API for drivers does not include any functionality to provide real-time clock access or append timestamps to frames. The summary of what needs to be done by the implementation while staying backwards compatible is as follows:

- Provide a way to enable timestamping on request;
- Return timestamp for a given TX frame to the PTP software;
- Return timestamps for RX frames to the PTP software;
- Provide access to the PTP clock;
- Provide means to update the clock;
- Provide means to update the frequency of the clock.

Chapter 5

PikeOS driver implementation

This chapter introduces the core ideas and a high-level overview of the implementation of the ENET_QOS controller driver. Critical design decisions and algorithms employed are presented.

5.1 Driver properties

Based on the hardware and PikeOS analysis, the following properties define the controller driver, further called *dwmac*:

- The driver is a user-level ExtFP DDK driver;
- It belongs to the NET DDK class;
- It uses the DDK Network High Level module;
- It implements the low-level API depicted in Listing 4.2;
- The driver has exclusive access to the controller;
- Generic PHY is accessed using the controller's MDIO registers.

Even though the driver is designed in a generic way over the platform, one notable exception currently limits its use to the MBa8MPxL target. PikeOS currently does not support its Clock Manager module on this platform. Clock Manager enables the configuration of the state and rate of clocks on a given platform. Because of this limitation, the driver currently includes the minimal code needed for this platform's ENET_QOS controller clock handling. The PikeOS team is familiar with this limitation.

Note that the following sections do not appear entirely in the order in which the driver implementation does all the necessary steps. They group logical blocks of operations the driver needs to do and handle. The reader may infer the order and dependencies of all these steps from the attached implementation source files.

5.2 Configuration

Before the controller can be set up, two parts of its configuration must be received. The driver configuration is split into the user PropFS configuration and configuration read from the controller.

5.2.1 PropFS configuration

The DDK NET logical and I/O device configuration was introduced in Section 4.2.1.4, and all properties were introduced in Tables 4.1 and 4.2.

One driver-specific boolean property extends the logical device configuration, enabling the reception of broadcast frames.

The second addition is the I/O device configuration from the target platform BSP. In this configuration, we must provide a value for a property of type `memmap`, which describes the address and size of the memory region where the controller registers are mapped on a given platform. For the MBa8MPxL target, the address is `0x30BF0000`, and the size is `0x2000` bytes. The second additional property is the interrupt identifier for interrupts coming from the controller. Because PikeOS uses an interrupt offset of 32, the value for our platform is 167.

Because PikeOS does not currently support its Clock Manager module on the target platform, we need two additional driver-specific properties in the BSP configuration. These are the default rates of clocks in Hertz driving the controller CSRs and PTP.

The final addition is a `uint32` property in the provider configuration describing the base address of the platform clock management registers. A value of 0 turns off any clock configuration by the driver. Clock Manager should replace this property.

All of these properties are summarized in Table 5.1.

Table 5.1 ENET_QOS driver specific configuration properties.

Pathname	Type	Decription	Default
		Driver	
`iores_clk`	uint32	Clock registers base.	0
		Logical device	
`bcast_enable`	bool	Broadcast reception enabled.	True
		BSP	
`iores0`	memmap	Memory map of CSR registers.	No default
`irq0`	uint32	Interrupt identifier.	No default
`clkhz_csr`	uint32	CSR clock rate.	266 666 666
`clkhz_ptp`	uint32	PTP clock rate.	125 000 000

5.2.2 Hardware configuration

The second part of the configuration comes from the hardware itself. This has two reasons. The controller is incrementally developed, and different revisions and types exist. The second reason is that the integrator may and must provide configuration parts using strapping pins on the controller itself. An example of such property is using either RMII or RGMII towards the PHY. The controller summarises this configuration in four registers, which must be read to set up the driver correctly. All properties the dwmac driver needs are listed in Table 5.2. Their usage is clarified further.

Table 5.2 ENET_QOS hardware configuration properties.

Name	Type	Decription
`maxspd`	uint32	Maximum enabled speed in Mbps.
`minspd`	uint32	Minimum enabled speed in Mbps.
`txqsiz`	uint32	MTL TX FIFO size in number of 256 byte blocks.
`rxqsiz`	uint32	MTL RX FIFO size in number of 256 byte blocks.
`hmapsiz`	uint16	Number of unicast and multicast filtering hash table entries.
`nvlans`	uint16	Number of perfect filtering VLAN entries.
`cnts`	bool	Diagnostic frame counters supported.
`mdio`	bool	MDIO access through controller enabled.
`hdx`	bool	Half duplex support enabled.
`tstamp`	bool	Timestamping support enabled. enabled.

5.3 Initialization

From the DDK framework's point of view, the driver initialization is done by calling the provider entry points in order of appearance as introduced in Listing 4.2. The high-level overview of the initialization of the dwmac driver is as follows:

1. **init_prov()** is called by the driver framework for the dwmac provider `eth0` where the driver initializes an instance of the framework's network services.

2. **init_io_dev()** is called for the driver's configured I/O device. Here, the driver first allocates its private data structure for the controller. The PropFS properties are read next. The I/O device is mapped into the driver's address space based on the `iores0` property. Interrupt handler based on the `irq0` property is attached with highest scheduling priority `DRV_THREAD_PRIO_HIGHEST`.

3. **init_logic_dev()** is called for the driver's logical device `dev0`. First, an instance of the High Level device is allocated. Then, all memory required by the controller descriptor rings is allocated. The PropFS configuration is verified. The

controller is then initialized without starting its operation. A thread for PHY configuration and link change events is started with the lowest scheduling priority `DRV_THREAD_PRIO_LOWEST`. Only after successful initialization the logical device entry points from Listing 4.2 are registered to make the controller available.

4. **init_complete()** is called to print diagnostic information about successful controller initialization to the serial console.

Any errors encountered during initialization are propagated, and it is up to the High Level module how it reacts.

5.3.1 Memory allocation

Dwmac must preallocate three memory regions. Because some of these are directly shared with the hardware through DMA access, careful cache management, memory operation order, and alignment must be considered. A notable feature of the PikeOS allocator is that it is a one-way allocator. This means the driver does not have any means to free allocated memory. This is done to simplify PikeOS certification.

The first region is the hardware descriptor ring, which spans from 4 to up to 1024 descriptors. The size of this ring is statically set to 512 based on other PikeOS drivers upon discussion with the PikeOS team. This is done for the RX and TX directions, which are allocated simultaneously. The TX ring starts at descriptor offset 0, and the RX ring follows at offset 512. The descriptor structure is depicted in Listing 5.1.

```
struct dma_descriptor {
    P4_uint32_t des0;
    P4_uint32_t des1;
    P4_uint32_t des2;
    P4_uint32_t des3;
};
```

■ **Code listing 5.1** ENET_QOS controller DMA descriptor. The four `uint32` fields represent bitfields, defined based on the direction, TX or RX, and based on the owner, driver or controller.

The allocation is done with memory page size alignment to satisfy the DMA bus alignment requirement. Because access permissions on this region must be set, the allocation size is rounded up to page size. First, the data cache of the allocated region is flushed, and then the access mode is changed to uncached strongly-ordered. This ensures the hardware sees our writes, and we can read back the data it writes without data loss. If this is not done, it could lead to data loss on CPUs with cache lines larger than one descriptor. When receiving a frame, the driver immediately queues the descriptor for reuse. This means the CPU needs to flush the cache line. If any other descriptor has been written by the hardware with another frame, that frame gets lost.

The second region is the frame data buffers. The High Level module preallocates these in a pool of structures called `mbuf` and handles the alignment. However, during frame reception, the buffer's data cache must be invalidated to force retrieval of the actual data stored in the buffer. On transmission, the buffer's data cache must be flushed for the same reason from the hardware's point of view.

Because dwmac does not handle the buffers, a pointer ring must be allocated to map the descriptors to the data buffers. The driver asks for the `mbuf` buffers from the High Level module, which returns pointers to them. No special handling of this memory region is needed.

5.3.2 Controller initialization

The controller initialization configures its three main blocks: the DMA, MTL and MAC. The register size of the controller is 32 bits. The framework provides the driver with `drv_io_write32()` and `drv_io_read32()` functions to access the controller registers. These operate on an instance of `struct drv_io_str`, which represents a mapped I/O device the driver receives from the framework during the `init_io_dev()` procedure. This is important because the framework initializes this structure with the platform endianness, and the driver is thus portable without any modifications or conditional compilation. Listing 5.2 shows an example of register modification using this API.

```
P4_uint32_t reg = drv_io_read32(&io, REGISTER_OFFSET)
                | REGISTER_FLAG;
drv_io_write32(&io, REGISTER_OFFSET, reg);
```

■ **Code listing 5.2** Usage of DDK platform independent register access.

The summarized order of controller initialization is as follows:

1. The unicast MAC address set in the controller is read. It may be set by the bootloader. This address is used if the High Level module sets no other address.
2. Software reset of the controller is done with a timeout of one second.
3. The hardware configuration introduced in Table 5.2 is read after reset.
4. The driver descriptor ring handling is initialized. Each RX descriptor is associated with a data buffer.
5. The DMA and MAC blocks are initialized without starting the frame reception and transmission. The MTL block is set to store-and-forward mode.
6. The PHY connected to the controller is autodetected and initialized.

5.3.3 DMA initialization

The DMA initialization consists of the descriptor rings and the AXI bus.

5.3.3.1 Descriptor rings

The descriptor is a simple structure of four 32-bit wide bitfields numbered 0 to 3, as shown in Listing 5.1. Its four bitfields differ based on the descriptor direction, that is, controller-to-driver or driver-to-controller, and by the type of operation, TX or RX. Bit 31 of field 3 is common to all of these modes. If this bit is set, the controller may use this descriptor. For the driver-to-controller direction, fields `des0` and `des1` hold the physical address of the `mbuf` associated with this descriptor.

For each direction, the controller descriptor ring gets configured with the address of the first descriptor, the number of descriptors in the ring and a tail pointer. The tail pointer indicates the first descriptor the driver owns. The controller consumes descriptors until the current descriptor pointer is less than the tail pointer. Then, the DMA goes into suspend mode and waits for a write to the tail pointer. The controller automatically wraps the address when the end of the ring is reached. The application must own at least one descriptor to differentiate between all descriptors being available and no descriptor being available. This operation is depicted in Figure 5.1. Dwmac provides the controller with 511 RX descriptors and no TX descriptors at initialization time.

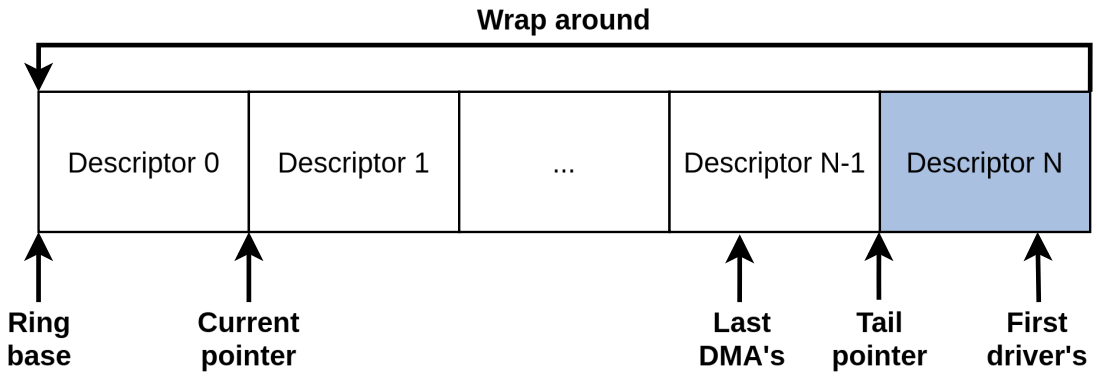


Figure 5.1 ENET_QOS DMA ring structure. Illustrated is one DMA ring. The blue descriptors belong to the driver; the white ones belong to the controller's DMA. The tail pointer indicates the first driver-owned descriptor. The current pointer indicates the controller's descriptor for the next frame. [32]

When working with the DMA descriptors, the platform endianness must be considered. The 32-bit wide bitfields are handled as little-endian integers by the controller. The driver framework provides us with the `drv_ltoh32()` and `drv_htol32()` functions, which do conversion between the host byte order and little-endian order. These must be used for all accesses to the descriptors.

5.3.3.2 AXI configuration

The AXI bus is based on transactions. We must configure the number of outstanding requests for read and write operations. This value is 16 for the MBa8MPxL. We do not support the AXI Low Power Interface (LPI), so it needs to be disabled. Also, read and write operations may be done in bursts to increase throughput. The maximum length of each burst in the number of beats must be configured. A beat is a unit of one read or write operation, 128 bits on the MBa8MPxL. At most, the size is limited by half of the MTL FIFO set in `txqsiz` and `rxqsiz` hardware properties. The last configuration property of the AXI configuration is the frame data buffer size in bytes. [32]

5.3.4 MAC initialization

The MAC block initialization is needed for proper frame filtering and PHY support.

First, the bootloader MAC address is written back to the controller, which was cleared by its reset. Based on the PropFS configuration, the filtering is prepared. As introduced in Section 4.4.4, the controller supports perfect and hash-based destination MAC address filtering on frame reception. Because different controller revisions do not expose the number of perfect MAC addresses allowed for perfect filtering in registers, only one address is filtered perfectly by dwmac, and the hash table handles the remainder. Therefore, we must configure the controller to pass frames that pass at least one of these filters. Broadcast frames are filtered based on the `bcast_enabled` property.

The configured frame MTS must be set in the MAC block in addition to the DMA block. By default, Watchdog and Jabber controller functions monitor the number of bytes on both frame paths and stop the transfer if more than 2048 bytes are detected. Because the user-configurable value is limited by 16 379 bytes, we must turn off these functions if the user requests more than 2048 bytes.

5.4 Frame transfers

Frame reception and transmission are turned off by default in the dwmac driver. Both are enabled on demand using the `start()` and `stop()` logical device callbacks. The operations done by the `start()` callback are:

1. Interrupts for the DMA channels get enabled.
2. The controller DMA RX and TX channels get enabled.
3. The MAC transmitter and receiver are started.
4. The High Level module frame queue is woken up.

The operations done by the `stop()` callback are:

1. The High Level module queue is stopped.
2. The controller DMA TX channel is stopped.
3. The driver waits for the MTL layer to finish all its transmissions to prevent errors on subsequent calls to `start()`.
4. The MAC transmitter and receiver are stopped.
5. The DMA RX channel is stopped.
6. The driver waits for all frame data to be transferred to the system memory to prevent data loss.
7. The DMA interrupts get disabled.

5.4.1 Frame interrupts

The only interrupts the dwmac driver enables are the ones for the DMA channel 0. As was described in Section 4.4.1.4, care must be taken not to miss any interrupts. The DMA controller exposes a global interrupt status register with a status bit for each enabled DMA channel. This register is self-cleared once all channel interrupts are handled. Each DMA channel has its interrupt status register. Only enabled interrupt events trigger changes in this register. The dwmac driver enables three events: frame received, frame transmission completed and bus error. This register must be cleared by writing 1 to the bit of the event.

The interrupt handling algorithm, depicted by a pseudo implementation in Listing 5.3, is as follows. Read the global status register. If the channel 0 status bit is not set, stop. Otherwise, read the channel 0 status register. Clear all bits in the channel 0 status register. If a bus error is detected, raise driver panic. If a transmission is finished, handle the TX ring update. If a frame was received, handle the RX ring update. Go back to reading the global status. Pseudo implementation is depicted in Listing 5.3.

5.4.2 Frame reception

Frame reception is handled solely by the RX interrupt callback. Once the controller marks a descriptor as belonging to the driver by clearing its bit 31 in the `des3` field, the descriptor's `mbuf` containing the frame data is returned to the High Level module. The driver remembers the index of the descriptor it should use for reception next. This indexing maps the given descriptor to its `mbuf`, which should be returned to the High Level model's queue.

```
dma = get_dma_status()
while dma is not empty:
    ch0 = get_status(channel=0)
    confirm_status(channel=0)

    if ch0 encountered DMA error:
        panic()
    if ch0 finised transmission:
        complete_tx(channel=0)
    if ch0 received frame:
        complete_rx(channel=0)

    dma = get_dma_status()
```

Code listing 5.3 Interrupt handling pseudocode of the dwmac driver.

The frame reception algorithm, depicted by a pseudo implementation in Listing 5.4, is as follows. Read the descriptor at the index. Read the ownership bit 31 of field 3. If the controller is the owner, stop. Otherwise, check the descriptor error bit. If it is set, queue the descriptor back. Set the owner bit, forward the driver index and set the DMA tail pointer to the address of this descriptor. Go back to reading the next descriptor. If no error is detected, request a new data buffer from the High Level module software queue. Drop the frame and queue the descriptor if no other buffer is available. Go back to reading the next descriptor. Otherwise, invalidate the data cache of the RX buffer and pass it to the High Level module software queue. Set the physical address of the new buffer in the descriptor. Flush the data cache of the new data buffer and queue the descriptor back. Go back to reading the next descriptor. Stop if the number of iterations reaches the length of the DMA ring.

5.4.3 Frame transmission

Frame transmissions are split into two parts. The driver remembers two indexes: the first descriptor it should use for the subsequent transmission and the oldest descriptor used for transmission. It also remembers the number of empty descriptors.

The first part involves enqueuing the frame into the hardware TX ring. This is done in the `transmit()` logical device callback. First, the passed-in data buffer length is checked against the configured MTS. If the buffer is too large, the frame is dropped. Next, the number of empty descriptors is checked. If no descriptor is available, the frame is dropped. Otherwise the data cache of the buffer is flushed. The physical address of the buffer is set in the descriptor, and the descriptor is queued the same way as in receive. The number of available descriptors is decremented. The High Level module software queue is stopped to prevent new frame transmissions if no other descriptor is available. On multicore systems, `transmit()` may still receive frames even if this is done.

The second part happens in the TX interrupt. Its handler walks the descriptor ring from the oldest index until a descriptor owned by the controller is reached. All `mbufs` associated with the descriptors until this one are returned to the High Level module for reuse. This loop also stops if the number of free descriptors equals the length of the ring. If the software queue was stopped by `transmit()`, it is enabled again. That is because if the interrupt was called, at least one descriptor must be available.

```
foreach i = 0..DMA_RING_LENGTH:
    des = dwmac.rx_des[dwmac.rxi]
    mbuf = dwmac.rx_mbuf[dwmac.rxi]

    if controller owns des:
        return

    if des contains error:
        set_des_own_bit(des)
        dwmac.rxi += 1
        dwmac.rxi %= DMA_RING_LENGTH
        set_dma_rx_tail(des)
        continue

    new_mbuf = alloc_mbuf()
    if got new_mbuf:
        invalidate_cache(mbuf)
        hlnet_rx(mbuf)
        mbuf = new_mbuf

    flush_cache(mbuf)
    dwmac.rx_mbuf[dwmac.rxi] = mbuf
    set_des_own_bit(des)
    dwmac.rxi += 1
    dwmac.rxi %= DMA_RING_LENGTH
    set_dma_rx_tail(des)
```

Code listing 5.4 RX interrupt handling pseudocode of the dwmac driver.

5.5 Filtering

Perfect unicast MAC address filtering is always enabled and is done by programming the MAC address bytes into the controller MAC address registers. The controller is assigned this address as its own.

5.5.1 MAC filtering

Hash-based filtering is enabled on demand separately for unicast and multicast traffic. In addition to enabling the traffic type, addresses that should pass must be programmed

into the filtering hash table. Programming of the table must be done carefully due to two reasons. The first reason for this is the fact that this table is shared between both kinds of traffic. The second is the PikeOS logical device API, which splits the configuration of unicast and multicast addresses into the `set_mac()` and `set_mcast()` callbacks. These callbacks receive as an input argument a list of MAC addresses the hash-based filter should pass for that kind of traffic. When any of these is called twice, the filter must be configured as if the first call did not happen. When an empty table is passed, the hash-based filter should be disabled for the given traffic type.

The table is represented by $N = \texttt{hmapsiz}/32$ controller registers, where 32 is the register size in bits. This implies $X = 32 - \log_2 \texttt{hmapsiz}$ bits are needed to index the entire table, of which 5 bits are needed to index into one register. Indexing is done by applying the IEEE 802.3 FCS algorithm to the MAC address. The result gets bit-reversed. That is, `1010` in binary becomes `0101`. This reversed value is shifted right by X bits, giving the `index`. The driver thus uses:

$$\texttt{register} = \texttt{index} \gg 5 \tag{5.1}$$

$$\texttt{bucket} = \texttt{index} \,\&\, \texttt{0x1F} \tag{5.2}$$

to enable filtering passthrough for each requested MAC address.

Let us consider an example. Both types of traffic have hash-based filtering enabled, but only multicast traffic enabled bits 2-63. Multicast filtering gets turned off by a call to `set_mcast()` with an empty list. If we only turn off multicast hash filtering by clearing its flag, unicast addresses hitting bits 2-63 will still get passed. There is no correctness problem because the software must handle the imperfect filtering. However, leaving the table like this introduces an unnecessary performance penalty. The same example can be applied to reprogramming the traffic type with a new address list.

The dwmac driver solves this by holding a software copy of the hash table for each traffic type separately. When any type gets reprogrammed, the table is configured only for the other type first, and the current type gets added to it afterwards. If the hardware property `hmapsiz` indicates no hash filtering support, multicast traffic is passed without filtering using the pass-all multicast flag. Unicast traffic is passed using the promiscuous mode flag in this case. That is, all traffic gets passed.

5.5.2 VLAN filtering

Even though PikeOS supports VLAN tagging on virtual devices, during dwmac implementation, it was discovered that no other project needed to implement hardware support for VLAN tagging. It was done only in software. Consequently, there exists no API to apply VLAN filters to Ethernet controllers.

The dwmac driver implements hardware support for VLAN filtering of single-tagged frames. It does not use the controller's more advanced features, such as applying or changing the frame tags, to stay compatible with the existing software implementation. The filtering may be done imperfectly, as with multiple MAC addresses, so the High Level module can decide if it wants to filter perfectly. The proposed C API is based on the MAC address API as follows:

```
P4_e_t set_vlan(
    P4_uint16_t eth_type,
    P4_uint16_t *vid_list,
    P4_size_t entries);
```

Code listing 5.5 Proposed PikeOS API for hardware VLAN filtering configuration.

The `set_vlan` function is given a list of the complete 16-bit VLAN tags, not just the 12-bit identifiers, and a single Ethernet frame type that applies to all of them. Thus, 801.Q and 802.1ad cannot be mixed. This is done because the same holds for the configuration of Ethernet devices through PropFS.

Dwmac configures the controller's VLAN filter registers based on the `nvlans` property. The VLAN promiscuous mode is enabled if the requested number of VLANs exceeds the number the hardware supports. The filtering is done based on the 12-bit identifiers of the given tags because the current software implementation does the same.

5.6 PHY control

The PHY implementation is done as a generic driver for any Clause 22 adhering PHY. This is done to make the dwmac driver portable as much as possible. The order of PHY initialization is as follows:

1. The MII bus is scanned on all 32 addresses starting with address 0 by a register read to autodetect the PHY. If multiple PHYs occupy the same bus, the PHY with the lowest address is used.

2. The PHY is reset using the Clause 22 Control register.

3. The PHY abilities get discovered. This includes detecting whether the PHY supports auto-negotiation, the link speed and duplex operation range, and whether the extended register set is supported.

4. The driver I/O device link configuration is checked against the PHY abilities. Without the extended register set, an I/O device link other than auto-negotiation gets refused as unsupported by the PHY. Because auto-negotiation is enforced for link speeds greater or equal to 1000 Mbps by IEEE 802.3 [34], the configuration

is refused if the PHY does not support the extended register set and the configured device link is set to 1000 Mbps. That is because the advertised speeds need to be changed to reflect this, which is impossible without the extended set.

5. The PHY link event thread is started.

5.6.1 Controller MDIO access

The Clause 22 protocol introduced in Section 4.4.2.2 is accessible through the controller's MDIO address and data registers. That is if the `mdio` hardware property is set. It is entirely transparent to the driver. To do any operation, the driver fills the address register with the PHY address, register address, and rate of the CSR clock. The rate is given to the driver in the `clkhz_csr` BSP property. The controller must select an appropriate divider because the MDIO clock rate cannot exceed 2.5 MHz and is driven from the CSR clock. The data to be read or written come from the MDIO data register.

These operations are started by setting the bit 0 of the address register and waiting for it to become 0. The dwmac driver implements these as blocking operations with a polling rate of 10 μs and a timeout of 1 ms. Therefore, any PHY operation may timeout.

5.6.2 Link event thread

A worker thread is started to configure the PHY, handle link change events and set the controller and RGMII TX clock. The polling rate is set to four times a second. Each iteration's current link state retrieved from the PHY is checked against the previous one. If the link changes to down, only a diagnostic message is shown. Even though PHYs should restart the auto-negotiation automatically, the driver reconfigures the PHY to prevent issues with non-conforming PHYs. If the link comes up, the controller and RGMII TX clock are configured for the negotiated link speed and duplex mode.

If auto-negotiation is used, the link speed must be resolved from the Clause 22 auto-negotiation advertisement registers. Full duplex mode is always given precedence over half duplex mode. First, the MASTER-SLAVE Control and Status registers 9 and 10 are examined. If both carry a flag for either 1000 Mbps full or half duplex mode, it is used as the negotiated link. Otherwise, the Auto-Negotiation Advertisement and Link Partner Base Page Ability registers 4 and 5 get examined. Any 100 Mbps mode gets precedence over any 10 Mbps mode.

5.6.2.1 Controller link configuration

When the link event thread encounters a new link-up event, the controller must be configured for the new settings. Nothing is done if the new link settings do not pass a check against the `maxspd`, `minspd` and `hdx` hardware properties.

First, the controller is stopped as if the `stop()` logical device callback was called. Modifying the MAC's configuration is not allowed if it is actively operating. Then, the duplex is changed because the MTL TX FIFO must be flushed when switching to half-duplex. Next, the controller's speed is changed. This is done by setting two bits toggling between 10, 100 or 1000 Mbps. Because RGMII is used on the MBa8MPxL, the RGMII TX clock towards the PHY must also be set to the correct frequency. This is the part of the driver that the Clock Manager should replace. The clock registers are not part of the ENET_QOS controller. The base frequency is 125 MHz, and a correct divider must be set based on the link speed. Table 5.3 lists all the divider settings. This is done only if the `iores_clk` property is not 0. The last step is starting the controller again if it was started before the link change event. The link thread ignores any errors as it does not have the means to react meaningfully. That is other than reporting the error.

Table 5.3 MBa8MPxL RGMII TX clock divider settings.

Mbps	Target MHz	Divider
1000	125	1
100	25	5
10	2.5	50

5.7 IOCTL interface

The low-level driver must provide three services through the `ioctl` interface.

The first service returns the current link state and speed. The returned value is the current link configuration set in the controller, which is saved when its configuration changes. If the controller's reconfiguration by the link event thread fails, this may be a different state than the PHY reports.

The second two services return and clear the state of the controller's diagnostic counters. The controller reports support for these in the `cnts` hardware property. If it is not set, the driver's only counters are drops counted by the `transmit()` callback and RX interrupt handler. If the property is set, the controller supports a granular set of 9 counters for the TX direction and 13 for the RX direction. These are combined with the drop counters kept by the driver. This combination is then mapped into a PikeOS counter structure for each direction depicted in Listing 5.6.

5.8 PTP integration

Implementing the functionalities needed for PTP integration on the MBa8MPxL target using the ENET_QOS controller is based on a previous experimental PikeOS integration of TSN on the ls1028a target.

```
struct drv_net_stats {
    P4_uint64_t bytes;
    P4_uint64_t frames;
    P4_uint64_t bcasts;
    P4_uint64_t mcasts;
    P4_uint64_t errors;
    P4_uint64_t drops;
    /* Only for TX direction. */
    P4_uint64_t collisions;
};
```

Code listing 5.6 PikeOS DDK NET diagnostic counters structure

5.8.1 PTP software

The PTP software running on the target is a modified PTPv2 implementation available on GitHub at `ptpd/ptpd`. It supports the E2E and P2P delay measurements while acting as an ordinary clock slave or master. It supports both the multicast and unicast transports over Ethernet and IP. It also supports the 802.1AS profile. It does not support the `Management` general messages. [38]

5.8.1.1 Clock servo

The central part of any PTP software is its clock servo algorithm. This component adjusts the local clock based on the `offsetFromMaster` and `meanDelay` values. It is the component PTP does not define. Figure 5.2 shows a block diagram of ptpd's clock servo.

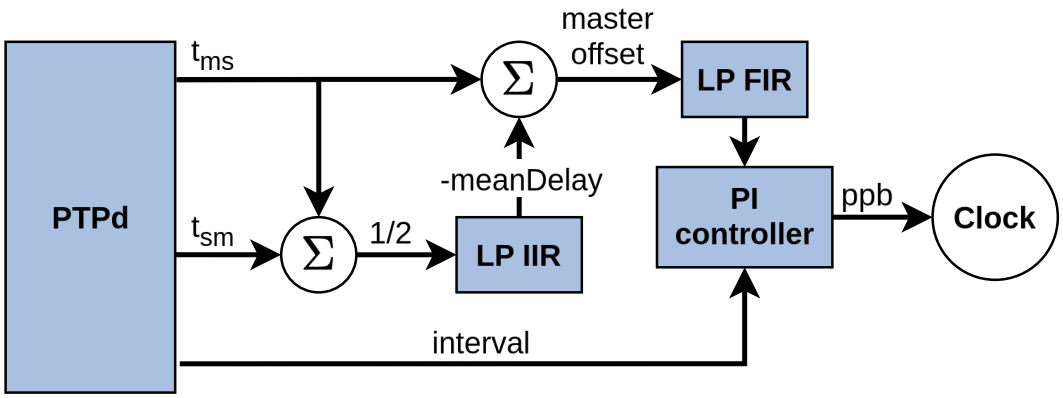


Figure 5.2 PTPd clock servo block diagram. The one-way delay is filtered using a low-pass IIR filter. The master offset is filtered using a two-sample average low-pass FIR filter. Output from the PI controller is the ppb adjust of the local clock. [39]

The output of ptpd's clock servo is the fractional tick-rate ppb adjustment of the local clock. It comes from its proportional-integral (PI) controller. The PI controller corrects both the frequency and time of the local clock. The proportional term is the

time difference between the master and slave clocks. The integral term is the frequency difference of the two clocks. Before the inputs are passed to the PI controller, they are filtered using two low-pass filters to reduce jitter. [39]

The master offset is filtered utilizing a finite impulse response (FIR) filter. It is implemented as a two-sample average:

$$y[n] = \frac{x[n]}{2} + \frac{x[n-1]}{2} \tag{5.3}$$

where $x[n]$ is the filter's input signal and $y[n]$ is the output signal. This is done because the PI controller does not filter effectively high-frequency noise. This filter introduces a one-sample delay, but the tracking error introduced by this is negligible. [39]

The one-way propagation delay passes through a first-order infinite impulse response (IIR) filter with a variable cutoff frequency defined as:

$$s \times y[n] - (s-1) \times y[n-1] = \frac{x[n]}{2} + \frac{x[n-1]}{2} \tag{5.4}$$

where s is the filter's *stiffness* term. This term controls the filter's phase and cutoff. It starts at a value of one, which leaves only the two-sample average, the same as the LP FIR. It then increments the stiffness term with each sample until reaching a maximum value. The higher s term lowers the filter's cutoff, which in turn smooths out the one-way delay. The one-way delay is filtered separately from the master offset for two reasons. [39]

As the one-way delay is sampled at a lower rate, it is interpolated in the combination with the master offset. The result of this interpolation is lowering the frequency of the one-way delay's noise. This allows more noise to pass through the LP filters. Separating the filtering eliminates the error introduced by the interpolation. [39]

Secondly, the one-way delay is dependent on the network topology. It thus has vastly different characteristics than the master offset. The one-way delay is nominally close to a constant. Therefore, it can be filtered using a low-cutoff filter without increasing the clock servo's tracking error. On the other hand, this property of ptpd's clock servo makes it less suitable for applications where the network topology is not stable. [39]

5.8.2 Clock control

The ENET_QOS clock control implementation provides the three functionalities required by ptpd: setting the system time, getting the system time, and adjusting the clock's frequency. Because ptpd returns ppb adjustments from its PI controller, it supports the fine system time update method.

A notable property of the implementation is that only the digital rollover mode is supported. The dwmac driver is the base for future integrations of similar controllers such as GMAC. This work does not delve into the implementation of other TSN features

because they cannot be implemented without the controller's PTP features first. However, future integrations may require features such as EST, and these are not supported by the controllers in the binary rollover mode [32].

5.8.2.1 DDK Real-time Clock Class

PikeOS tracks only monotonic time in units of nanoseconds since startup in a 64-bit unsigned integer `P4_time_t`. Its resolution depends on the platform and configuration. [6] To provide applications with access to wall time clocks, the PikeOS x86 architecture variant introduced the DDK Real-time Clock Class (RTC). This class is typically implemented as a Kernel Driver providing a file. The ls1028a integration adapted the RTC file API for PTP clocks:

- `start()` is called by the RTC framework when an application opens the driver's file for writing;
- `set_timestamp()` directly sets the clock's system time to the one provided in its `P4_time_t` parameter in the PikeOS format;
- `get_timestamp()` retrieves the clock's system time in the PikeOS format into its `P4_time_t` parameter;
- `adjust()` modifies the clock's frequency based on a 32-bit signed integer parameter carrying the ppb adjustment.

5.8.2.2 ENET_QOS PTP Kernel Driver

This API is implemented by a second driver, *dwmac_ptp* RTC Kernel Driver. The ptpd control servo accesses the ENET_QOS's PTP clock through this driver. The implementation ignores the controller's 16-bit higher-second register for simplicity, which limits the second resolution to 32 bits. That is to the year 2106.

The `set_timestamp()` function first transforms the PikeOS time to the controller's representation in split seconds and nanoseconds:

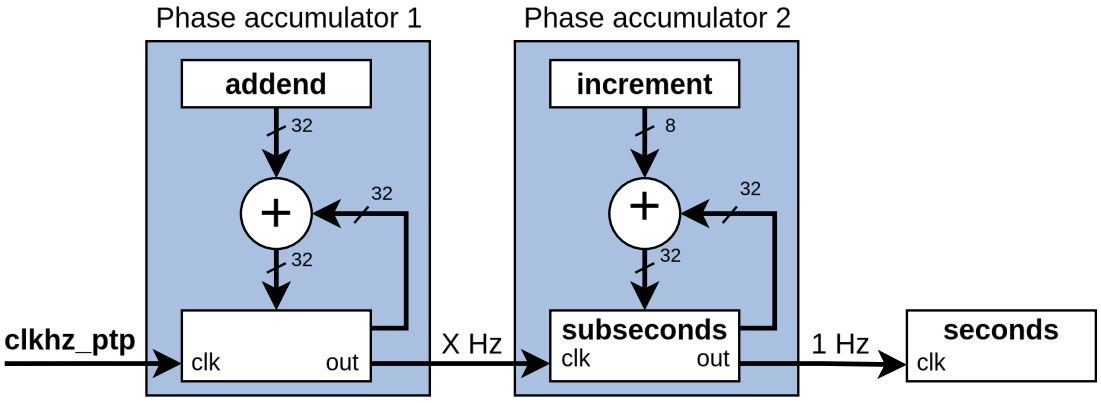
$$\texttt{seconds} = \frac{\texttt{P4\_time\_t}}{10^9} \tag{5.5}$$

$$\texttt{nanoseconds} = \texttt{P4\_time\_t} \ \% \ 10^9 \tag{5.6}$$

and sets its seconds and nanoseconds 32-bit registers to these values. The inverse is done by the `get_timestamp()` function:

$$\texttt{P4\_time\_t} = \left(\texttt{seconds} \times 10^9\right) + \texttt{nanoseconds} \tag{5.7}$$

The core function of the driver is the `start()` function. It first checks if the main driver enabled the timestamping block; if not, it returns an error. Otherwise, it initializes the values of the `increment` and `addend` registers. When the controller's timestamping block operates in fine mode, the timestamping and timekeeping employ a pair of phase accumulators [40]. This is depicted in Figure 5.3.



■ **Figure 5.3** ENET_QOS double phase accumulator clock. Depicted is the timestamping utilizing the addend register. Its value is added to a 32-bit accumulator, which ticks every time it overflows. On each overflow, the subsecond register is incremented by the configured amount. On each subsecond overflow, the seconds register is incremented. The subseconds register is depicted as 32-bit since it supports the digital and binary rollover modes. [40]

The seconds register of the clock is incremented on each overflow of the subsecond accumulator. Because the implementation considers only the digital rollover mode, this overflow occurs once the subsecond accumulator reaches 10^9 nanoseconds. The accumulator is incremented by an `increment` nanoseconds configured by the driver.

In the coarse update method, this is the period of the PTP oscillator, which has a frequency defined by the `clkhz_ptp` property. The subnanosecond remainder may also be configured, multiplied by 2^8. As an example, the nanosecond increment is five for 5.3 ns period, and the subnanosecond increment is `0x4C`. [32] The driver implements this in picoseconds to stay in integer arithmetics:

$$\texttt{ps} = \frac{10^{12}}{\texttt{clkhz\_ptp}} \tag{5.8}$$

$$\texttt{ns} = \frac{\texttt{ps}}{1000} \tag{5.9}$$

$$\texttt{subns} = \frac{(\texttt{ps} \ \% \ 1000) \times 256}{1000} \tag{5.10}$$

In the fine method, the subsecond increment is driven by a frequency generated by overflowing the addend accumulator. The PTP oscillator drives its increment. [32] Let y be the number of overflows of the subsecond accumulator in one second. If this value gets multiplied by 2^{32}, the size of the addend accumulator, the result represents the total

of the `addend` value added `clkhz_ptp` times. That is, `addend` is the total divided by the oscillator's rate. [40] Mathematically:

$$y = \frac{10^{12}}{\texttt{ps}} \tag{5.11}$$

$$\texttt{addend} = \frac{y \times 2^{32}}{\texttt{clkhz\_ptp}} \tag{5.12}$$

The driver doubles the `ps` value from Equation 5.8. This is done because the calculated `addend` is divided by two and thus equals 2^{31}. That is, if the remainder of the division by the oscillator rate is 0. This allows its fine-grained control in both directions by the ptpd's clock servo, which controls the frequency by adjusting the `addend` value. As an example, for a 125 MHz PTP oscillator, period of eight nanoseconds, the `ps` is 16 000, and the `addend` is `0x80000000`.

The frequency adjustments are done through the `adjust()` function, which receives a 32-bit signed ppb value. The driver then calculates the new `addend` value as:

$$\texttt{new} = \texttt{addend} + \frac{\texttt{addend} \times \texttt{ppb}}{10^9} \tag{5.13}$$

and writes it into the addend register. The calculation always uses the `addend` value from driver initialization. That is without any previous modifications.

5.8.3 Timestamping

Timestamping is enabled on demand by the application requesting timestamping using an `ioctl` call through the main dwmac driver. This is done to keep the drivers backwards compatible. Dwmac, in this case, configures the DMA rings for timestamping, initializes the timestamping and programs the correction values.

Drivers provide timestamps for PTP packets in the packet's `mbuf` buffer. On the RX data path, the 8 bytes of the timestamp are appended to the received packet data. On the TX path, the timestamp bytes are written at the beginning of the buffer. It is then up to the modified DDK NET High Level module to provide the timestamp further to the application. This depends on whether the new boolean `enable_ptp` property is set on the virtual device. If it is not, the module removes the timestamp from the buffer before passing it to the application.

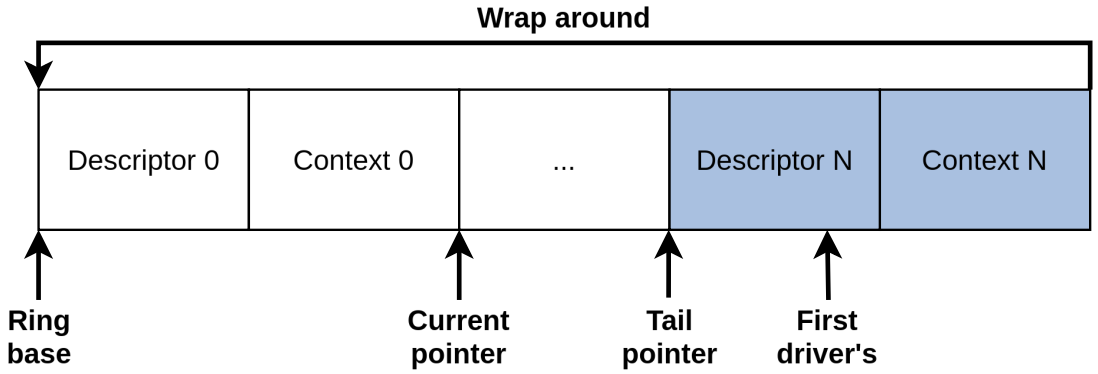
Packets which should get timestamped in the TX direction are issued through another `ioctl` call instead of the regular packet send method utilizing the `write` function. This interface was chosen because applications may selectively request only some TX packets to get timestamped. Furthermore, the `ioctl` call returns the timestamp for the given packet in the same context to the application.

5.8.3.1 DMA modifications

For both the TX and RX channels, a timestamp for a given packet is given in the DMA descriptor's fields `des0`, holding the nanoseconds, and `des1`, holding the seconds. These fields are transformed to the `P4_time_t` representation using Equation 5.7.

The driver requests TX timestamps by setting a timestamping bit in the packet's descriptor before passing it to the controller. The driver may request one-step timestamping, but the application API currently does not provide a means for the applications to request one. In the `transmit()` callback, the driver then waits on the given descriptor until the controller marks the descriptor as not owned by it. The driver then verifies that the descriptor contains no error bits set and retrieves the timestamp. This operation may timeout if no timestamp is received or the descriptor is marked again as belonging to the controller.

During the timestamping initialization, the driver modifies the RX DMA ring to retrieve the RX timestamps. The controller is configured to timestamp every received packet. It is then up to the High Level module and the application to decide if they are interested in the packet's timestamp. The controller returns the timestamp in an RX descriptor called the context descriptor. Its sole purpose is to return the timestamp of an RX packet. This descriptor is the descriptor following the packet's last descriptor. Because dwmac operates in store-and-forward mode, it is the packet's second descriptor. During the timestamping initialization, each second descriptor is cleared of its associated `mbuf` structure for its reuse. This, in turn, effectively cuts the RX descriptor ring length to half. That is 256 descriptor. The modified ring structure is depicted in Figure 5.4.



■ **Figure 5.4** ENET_QOS timestamping DMA ring structure. The blue descriptors belong to the driver; the white ones belong to the controller's DMA. Indicated is that every second descriptor is a context holding the packet's timestamp. [32]

5.8.3.2 Timestamp correction

In the ingress direction, the timestamp captured by the MAC is delayed to the point when the SOF of the incoming frame hits the local PHY. The captured timestamp must be thus reduced. The unknown PHY latency is not corrected because the driver expects operation with a generic PHY. The MAC's internal nanosecond latency is read from its latency status register. Because the correction value is added to the timestamp by the MAC, it must be programmed back with the bit 31 set for negative value and bits[30:0] set to $10^9 -$ `ingressLatency`. This is done on every link speed change.

In the egress direction, the internal latency is read from the egress latency status register and written back for correction without modification. These four combined read and write operations on the internal latencies must be done by drivers because the controller cannot automatically convert them to the correct format when operating in the binary rollover mode. [32]

The CDC synchronization error specified as $2 \times$ `clkhz_ptp` is added to the latency in the ingress direction and subtracted in the egress direction by the driver before programming the correction [32]. This can be done only because one-step TX timestamping is currently not supported. Otherwise, these corrections would have to be applied manually by the driver when reading the timestamps back because the equation for one-step mode differs, and this mode is enabled on demand per frame.

Chapter 6

PikeOS driver testing

This chapter is dedicated to testing all the implemented functionalities of the ENET_QOS Ethernet controller through the dwmac driver. The tests include basic functionalities, advanced filtering, and essential tests of the PTP implementation.

6.1 Setup

The driver is tested through a direct Ethernet connection between a Lenovo ThinkPad E14 Gen 4 AMD laptop and the MBa8MPxL target. This is done to exclude the influence of the network topology and configuration on the testing procedures. Furthermore, this simplifies the forging of testing packets because values of fields such as TTL can be ignored since no routers are engaged. If not specified otherwise, the driver binary used is the development compilation variant as provided by the PikeOS IDE. That means compiler optimizations are disabled, and assertions are enabled to catch invalid states.

The attached implementation includes test cases in Python and shell scripts. The only supported platform is Linux. Any distribution may be used under the condition that `bash`, `ethtool`, `ping`, `Python` and `Scapy` packages are available. Scapy is a Python library that enables decoding and forging packets of a wide range of protocols, sending network traffic and matching responses [41].

Table 6.1 lists configuration properties common to all tests.

6.2 Basic functionality

The basic functionality test verifies the core functionality of the driver, that is, packet send and receive. It utilizes an instance of an Internet Control Message Protocol (ICMP) echo server in PikeOS. The server is configured with one virtual interface. This ICMP interface is assigned the real device dwmac provides. It is assigned the link-local IP address `169.254.0.100`.

Table 6.1 PropFS configuration common to all tests.

Pathname	Default
I/O device	
aneg	True
Real device	
enable_mcast	True
mac_address	00:00:00:00:00:00
mbuf_pool_size	2048
mcast_table_size	32
receive_queue_depth	512
send_queue_depth	512
dwmac	
bcast_enable	True
clkhz_csr	266 666 666
iores_clk	0x30380000
iores0	0x30bf0000
irq0	167

This test includes verification of the `bcast_enable` property. Scapy allows sending and matching frames of the Address Resolution Protocol (ARP), which utilizes broadcast Ethernet traffic. With the property disabled, an ARP request for the target's MAC address is made using Scapy. The expected result is a timeout since the frame should not traverse through the controller. Then, the property is enabled, and the test is repeated. In case of an ARP reply, one ICMP `echo-request` is issued towards the target's IP address, and an `echo-reply` is expected to be received.

The test results, listed in Table 6.2, indicate that the driver correctly receives and transmits packets. They also show that it properly filters broadcast traffic based on the `bcast_enable` property.

Table 6.2 Results of the basic driver test.

Traffic type	Expected	Actual
`bcast_enable` = False		
ARP	No response	No response
ICMP	No response	No response
`bcast_enable` = True		
ARP	Response	Response
ICMP	Response	Response

6.3 MAC address filtering

The MAC address filtering test verifies the driver's configuration of the hash-based filtering. It utilizes an instance of the ICMP echo server configured with four virtual interfaces. Each ICMP interface is assigned one of the four virtual devices dwmac provides. Each interface is assigned an IP address in the form `169.254.0.10X` where X is the interface number starting at index 0. In addition, the logical device 0 has the `enable_mcast` property set. The matching ICMP interface is assigned the multicast address `224.0.0.1`. That is multicast MAC address `01:00:5e:00:01:2a`. Manual verification ensures that the multicast and virtual autogenerated MAC addresses correspond to five different hash table buckets.

The basic functionality test has verified that the driver correctly assigns the controller's perfect unicast MAC address. The unicast hash-based filter is tested first by turning off the code inside the `set_mac()` using conditional compilation block `#if 0` but still returning success. In this configuration, no interface should respond to an unicast ICMP echo. Then, the code is compiled back in, and the test is repeated, expecting all the interfaces to respond. The multicast test is executed similarly but, in addition, configures the `enable_mcast` property of the real device. Even without the body of the `set_mac()` callback, a response is expected in the multicast case because the filtering happens only on ingress from the external network, not the device itself.

The test results, listed in Table 6.3, demonstrate that the driver correctly configures the hash-based MAC address filter for each traffic type separately. That is because each assigned MAC address corresponds to a different bucket.

Table 6.3 Results of the hash-based MAC address filter test.

Traffic type	Expected	Actual
`set_mac()` with `#if 0`, `enable_mcast` = False		
ICMP unicast	No response	No response
ICMP multicast	No response	No response
`set_mac()` with `#if 0`, `enable_mcast` = True		
ICMP unicast	No response	No response
ICMP multicast	Response	Response
`set_mac()`, `enable_mcast` = False		
ICMP unicast	Response	Response
ICMP multicast	No response	No response
`set_mac()`, `enable_mcast` = True		
ICMP unicast	Response	Response
ICMP multicast	Response	Response

6.4 VLAN filtering

The VLAN filtering test verifies the proposed API for configuring the VLANs in Ethernet drivers utilizing the ICMP echo server. Since accessing the VLAN configuration is impossible, the call of `set_vlan()` function must be compiled in with static configuration. Both 802.1Q and 802.1ad are tested separately, ensuring the other type does not pass the configured filter. The real device is configured by setting the `vlan_dev` and the `vlan_tag_ethertype` based on the tagging mode. The virtual device's 0 VLAN ID is set in the `vlan_id` property. This configuration is reflected in the call to `set_vlan()`.

First, 802.1Q VLAN 100 is configured on the virtual device. Then, ICMP echo is issued on four Linux interfaces: 802.1Q 100, 802.1Q 200, 802.1ad 100 and 802.1ad 200. The reply should be received only on the first interface. Then, the same test is repeated with the virtual device configured as an 802.1ad VLAN 100. The expected outcome is a reply on the 802.1ad 100 Linux interface.

The ICMP results listed in Table 6.4 display that the driver properly configures the perfect VLAN filters in the controller.

Table 6.4 Results of the VLAN filtering test.

Linux interface	Expected	Actual
PikeOS 802.1Q 100		
100Q	Response	Response
200Q	No response	No response
100ad	No response	No response
200ad	No response	No response
PikeOS 802.1ad 100		
100Q	No response	No response
200Q	No response	No response
100ad	Response	Response
200ad	No response	No response

6.5 PHY link control

The PHY link control test examines multiple parts of the driver. The generic Clause 22 PHY code is tested. That is PHY configuration and autonegotiation result parsing. The controller link speed configuration and the `start()` and `stop()` callbacks are tested. The platform-dependent clock control code replacing the Clock Manager is tested.

The test is separated into two parts executed utilizing the ICMP echo server on the real device. First, the driver is configured to use autonegotiation, and the Linux side gets configured for all the six combinations of 10/100/1000 Mbps link speed and half/full-

duplex mode autonegotiation utilizing the `ethtool` utility. Then, both sides are assigned a static configuration, without autonegotiation, for each combination. The result is 12 link configurations, each tested by an ICMP echo when the link comes up.

The results in Table 6.4 indicate that the driver correctly configures the PHY and parses the negotiated link to configure the controller and the external RGMII TX clock.

Table 6.5 Results of the PHY control test.

Speed	Half duplex	Full duplex
10	Response	Response
10 auto	Response	Response
100	Response	Response
100 auto	Response	Response
1000	Response	Response
1000 auto	Response	Response

6.6 DMA load

The DMA load test verifies the handling of the DMA ring structure in both directions under heavy load. It ensures that the rings stay consistent and that no packet drops occur. This test is performed against one ICMP interface mapped to the real device. The `ping` utility on the Linux side is set in `-f -l512` mode. That is flood ping with a preload of 512 ICMP echo requests, which keeps 512 requests on the wire without waiting for a reply. This configuration maintains the DMA rings full in both directions because the dwmac configured ring length is 512. This is run continuously for four hours, which the team agreed upon as long enough. This test uses a binary compiled with optimizations and assertions enabled in the IDE build configuration.

The result is no packet drops over the whole period of the measurement. Repeating the test with a preload value of 513 exposes packet drops in units of seconds. That is expected behaviour because the DMA rings are full and cannot keep up with the number of transmitted frames.

6.7 PTP integration

The PTP integration is tested with the target sat in a laboratory, not in a direct connection with the Linux host. That is, multiple hops are made through non-PTP network nodes. This is done to replicate the setup used for the integration on the ls1028a target.

The master side is the `ptpd/ptpd` daemon, commit `#1ec9e65`, running on the Linux host in forced master mode with P2P link delay measurement. It is free-running (not synchronized with any upstream master) with the ARB timescale set. Real absolute

time is not crucial in determining the correctness of the dwmac implementation. The message rate is set to 0 for both the `Sync` and `Pdelay_Req` messages. The transport utilized is unicast UDP over IP. No specific PTP profile is used.

The samples of the offset from the master as measured by the target's `ptpd` instance are taken over one hour every second after the slave reaches calibration for the first time. The slave enters the calibrated state when its offset reaches 10 μs. The slave returns to the uncalibrated state if its offset from the master reaches 1 ms. [38]

The collected data, plotted in Figure 6.1, indicates functioning PTP implementation. The calculated resulting cTE is 0.716 μs, and max|TE| is 293.212 μs. This is in the expected range based on the experimental integration on the ls1028a target. That implementation reached cTE of 30 μs over a non-direct (multi-hop) LAN connection [42]. The difference may come from the underlying oscillator quality and network load.

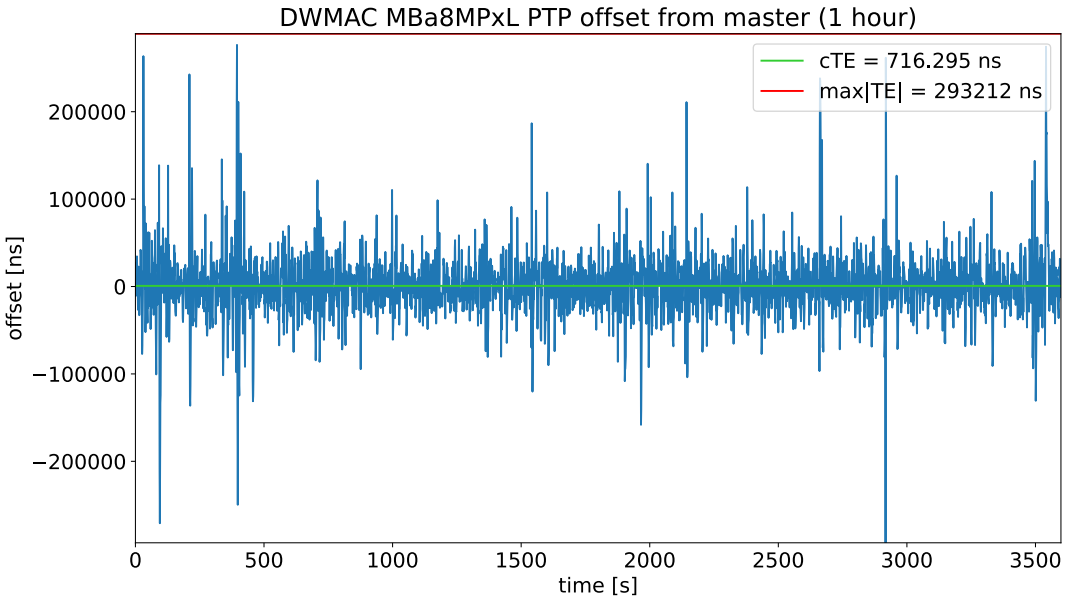


Figure 6.1 Offset from master plot of the PTP test. Plotted are TE samples taken each second over a 1-hour measurement against a free-running Linux master after the target reached the calibrated state. The measured cTE is 0.716 μs, and max|TE| is 293.212 μs.

The primary reason for not measuring MTIE and TDEV is that these metrics are typically used to determine the quality of the resulting clock. However, with the current experimental support, it is sufficient to ascertain if the integration is functioning with any clock quality. Moreover, since the same methodology was successfully applied to the ls1028a target, it is valuable to have comparable results.

The second reason is that only specialized network equipment can measure these values. For example, it must be synchronized to the GNSS and requires some form of clock output from the target, such as a PPS signal. With how the target is set up, that is not possible.

Chapter 7

Conclusion

The main objective of this thesis was to study time-sensitive networking features required to integrate TSN into the PikeOS real-time operating system and experimentally implement a selected subset on the TQ Systems MBa8MPxL SBC.

The analytical part of the thesis first introduced the concept of real-time operating systems. The base terminology was introduced to emphasize the differentiation of operating systems kernels. The PikeOS RTOS was introduced as an exemplary real-time microkernel. Following that, time-sensitive networking was explored. The focus was placed on time synchronization in TSN, which is required for all other TSN functionalities. It was studied from the definition of time, how it is kept using clocks and how their quality is measured. Emphasis was set on packet-based time synchronization methods employing the Precisisin Time Protocol in TSN. PTP's central mathematical synchronization model, algorithms and involved entities were analyzed.

The implementation part of the thesis first explored the PikeOS driver model. It studied the classes of PikeOS drivers and how they are configured. An in-depth analysis of the network class was done, including its High Level module, which abstracts the hardware-independent part of Ethernet drivers. A low-level driver for a new Ethernet controller, ENET_QOS, was designed and implemented. Its feature set included an integrated driver for any IEEE Clause 22 conforming PHY and advanced features such as VLAN filtering, which required a new standard API proposal. Building on a previous TSN experiment on the PikeOS ls1028a target, PTP features of the controller were enumerated as a selected TSN subset to integrate. These were implemented utilizing a new experimental API for network drivers and required a third driver.

Testing the driver asserted the implementation correct. All the driver's functionalities were tested. These included basic packet send and receive, unicast traffic filtering, and advanced multicast and VLAN-tagged traffic filtering. The PHY driver was tested against the SBC's Texas Instruments PHY. The PTP test against a Linux master measured an offset in the expected range of tens of microseconds.

The implementation will be used in a real project utilizing a derivative controller called GMAC. That application will require other TSN features, such as the 802.1Qbv protocol. The central focus of future work building on this thesis will be the integration of this and other TSN protocols utilizing the PTP synchronized time base, which was accomplished by this thesis.

Bibliography

1. WANG, Jiacun. *Real-Time Embedded Systems*. 1st edition. John Wiley & Sons, Inc., 2017. ISBN 978-1-118-11617-3.
2. SILBERSCHATZ, Abraham; GALVIN, Peter; GAGNE, Greg. *Operating system concepts*. 10th edition. John Wiley & Sons, Inc, 2018. ISBN 978-1-119-32091-3.
3. DOEPPNER, Thomas. *Operating systems in depth*. 1st edition. John Wiley & Sons, Inc., 2010. ISBN 978-0-471-68723-8.
4. SINGH, Samesun; TORRI, Stephen. Microkernel operating systems compared to monolithic operating systems: a review on functional safety [online]. 2023 [visited on 2024-03-24]. Available from: `https://www.researchgate.net/publication/376582652_Microkernel_operating_systems_compared_to_monolithic_operating_systems_a_review_on_functional_safety`.
5. APPLE-OSS-DISTRIBUTIONS. *xnu* [online]. GitHub, 2023 [visited on 2024-03-25]. Available from: `https://github.com/apple-oss-distributions/xnu`.
6. SYSGO GMBH. *PikeOS User Manual*. 2023. No. 5.1-1103.
7. ZURAWSKI, Richard. *Industrial Communication Technology Handbook*. 2nd edition. CRC Press, 2017. ISBN 978-1138071810.
8. PINDORIA, Amit. *Understand the IEEE standard creation and ratification process and identify IEEE standard naming conventions – Dot11AP* [online]. 2018. [visited on 2024-03-29]. Available from: `https://dot11ap.wordpress.com/understand-the-ieee-standard-creation-and-ratification-process-and-identify-ieee-standard-naming-conventions/`.
9. JOHAS TEENER, Michael D.; FREDETTE, Andre N.; BOIGER, Christian; KLEIN, Philippe; GUNTHER, Craig; OLSEN, David; STANTON, Kevin. Heterogeneous Networks for Audio and Video: Using IEEE 802.1 Audio Video Bridging. *Proceedings of the IEEE*. 2013, vol. 101, no. 11, pp. 2339–2354. Available from DOI: `10.1109/JPROC.2013.2275160`.

10. BROWEN, Carrie. *Why use 10BASE-T1S instead of CAN? | Keysight Blogs* [online]. 2024. [visited on 2024-03-29]. Available from: https://www.keysight.com/blogs/en/tech/2024/02/8/how-is-10base-t1s-different-from-can.

11. YOKOYAMA, Takanori; MATSUBARA, Ayane; YOO, Myungryun. A Real-Time Operating System with GNSS-Based Tick Synchronization. In: *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*. 2015, pp. 19–24. Available from DOI: 10.1109/CPSNA.2015.13.

12. HAGARTY, Dennis; AJMERI, Shadid; TANWAR, Anshul. *Synchronizing 5G Mobile Networks*. 1st edition. Cisco Press, 2021. ISBN 978-0-13-683625-4.

13. INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC. *Time-Sensitive Networking (TSN) Task Group* [online]. 2024. [visited on 2024-03-29]. Available from: https://1.ieee802.org/tsn/.

14. INTERNATIONAL TELECOMMUNICATION UNION. Definitions and terminology for synchronization networks. *ITU-T-G.810* [online]. 1996, no. E 9794, pp. 1–27 [visited on 2024-03-30]. Available from: https://www.itu.int/rec/T-REC-G.810-199608-I/en.

15. BUREAU INTERNATIONAL DES POIDS ET MESURES. - *second - BIPM* [online]. [visited on 2024-04-04]. Available from: https://www.bipm.org/en/si-base-units/second.

16. INTERNATIONAL TELECOMMUNICATION UNION. *Figure 1 – MTIE as a function of an observation (integration) period* [online]. 2018-11-29. [visited on 2024-03-30]. No. ITU-T-G.8272. Available from: https://www.itu.int/rec/T-REC-G.8272-201811-I/en.

17. INTERNATIONAL TELECOMMUNICATION UNION. Timing characteristics of primary reference clocks. *ITU-T-G.811* [online]. 1997, no. E 12242, pp. 1–27 [visited on 2024-03-30]. Available from: https://www.itu.int/rec/T-REC-G.811-199709-I/en.

18. INTERNATIONAL TELECOMMUNICATION UNION. *Figure 2 – TDEV as a function of an observation (integration) period* [online]. 2018-11-29. [visited on 2024-03-30]. No. ITU-T-G.8272. Available from: https://www.itu.int/rec/T-REC-G.8272-201811-I/en.

19. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)*. 2020, pp. 1–499. Available from DOI: 10.1109/IEEESTD.2020.9120376.

20. MILLS, D.L. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*. 1991, vol. 39, no. 10, pp. 1482–1493. Available from DOI: 10.1109/26.103043.

21. NETWORK TIME FOUNDATION. *Clock Discipline Algorithm* [online]. 2014-03-10. [visited on 2024-04-10]. Available from: https://www.eecis.udel.edu/~mills/ntp/html/discipline.html.

22. NETWORK TIME FOUNDATION. *5. How does it work?* [online]. 2022-06-27. [visited on 2024-04-10]. Available from: https://www.ntp.org/ntpfaq/ntp-s-algo/#513-performance.

23. INTERNET ASSIGNED NUMBERS AUTHORITY. *Service Name and Transport Protocol Port Number Registry* [online]. 2024-03-18. [visited on 2024-04-15]. Available from: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

24. NETTIMELOGIC GMBH. *PTP Basics* [online]. 2022. [visited on 2024-04-15]. Available from: https://www.nettimelogic.com/resources/PTP%20Basics.pdf.

25. GARNER, Geoffrey M. *Use of IEEE 1588 Best Master Clock Algorithm in IEEE 802.1AS* [online]. 2022-06-27. [visited on 2024-04-15]. Available from: https://www.ieee802.org/1/files/public/docs2006/as-garner-use-of-bmc-061114.pdf.

26. IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications. *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)*. 2020, pp. 1–421. Available from DOI: 10.1109/IEEESTD.2020.9121845.

27. CERN. *White Rabbit Official CERN website* [online]. 2024-02-19. [visited on 2024-04-15]. Available from: https://white-rabbit.web.cern.ch/.

28. SYSGO GMBH. *PikeOS Device Driver Programming Reference Manual*. 2023. No. 5.1-311.

29. TQ-SYSTEMS GMBH. *Embedded Module MBa8MPxL | TQ* [online]. 2023-03-16. [visited on 2024-02-24]. Available from: https://www.tq-group.com/en/products/tq-embedded/arm-architecture/mba8mpxl/.

30. NXP SEMICONDUCTORS. *i.MX 8M Plus | Cortex-A53/M7 | NXP Semiconductors* [online]. 2021-01-12. [visited on 2024-02-24]. Available from: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/i-mx-applications-processors/i-mx-8-applications-processors/i-mx-8m-plus-arm-cortex-a53-machine-learning-vision-multimedia-and-industrial-iot:IMX8MPLUS.

31. TQ-SYSTEMS GMBH. *EMB MBa8MPxL | TQ* [online]. 2023-03-16. [visited on 2024-02-24]. Available from: https://www.tq-group.com/fileadmin/assets/products/embedded/Arm/EMB_MBa8MPxL.jpg.

32. NXP SEMICONDUCTORS. i.MX 8M Plus Applications Processor Reference Manual [online]. 2021, vol. 1, pp. 3957–4979 [visited on 2024-02-25]. Available from: https://www.nxp.com/webapp/Download?colCode=IMX8MPIEC.

33. NXP SEMICONDUCTORS. i.MX 8M Plus Applications Processor Reference Manual Preview [online]. 2021, vol. 1, pp. 1–11 [visited on 2024-02-25]. Available from: https://www.nxp.com/docs/en/preview/PREVIEW_IMX8MPIEC.pdf.

34. IEEE Standard for Ethernet. *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*. 2016, pp. 1–4017. Available from DOI: 10.1109/IEEESTD.2016.7428776.

35. INTEL CORPORATION. *5.1.7.1.2. RMII and RGMII PHY Interfaces* [online]. Intel Corporation, 2023-10-09 [visited on 2024-02-25]. Available from: https://www.intel.com/content/www/us/en/docs/programmable/683634/21-2/rmii-and-rgmii-phy-interfaces.html.

36. INTEL CORPORATION. *5.1.7.1.5. MDIO* [online]. Intel Corporation, 2023-10-09 [visited on 2024-02-26]. Available from: https://www.intel.com/content/www/us/en/docs/programmable/683634/21-2/mdio.html.

37. IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks. *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*. 2018, pp. 1–1993. Available from DOI: 10.1109/IEEESTD.2018.8403927.

38. PTPD. *ptpd* [online]. GitHub, 2019 [visited on 2024-04-20]. Available from: https://github.com/ptpd/ptpd.

39. CORRELL, Kendall; BARENDT, Nick. Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol [online]. 2006 [visited on 2024-04-20]. Available from: https://www.researchgate.net/publication/236213185_Design_Considerations_for_Software_Only_Implementations_of_the_IEEE_1588_Precision_Time_Protocol.

40. BREUER, Jan. *Synchronizace času v distribuovaných heterogenních měřicích a řídicích systémech*. Prague, 2016. Dissertation thesis. Czech Technical University in Prague.

41. SCAPY COMMUNITY. *Scapy* [online]. 2024. [visited on 2024-04-26]. Available from: https://scapy.net.

42. KLIMMEK, Matthias. *Architecture and Study of TSN on ls1028a target*. 2023.

Appendix A

Compilation and testing

Codeo IDE must be used to compile the drivers. Import both the `dwmac` and `dwmac_ptp` projects using the `File -> Import` function. Then, both can be compiled using the `all` build target. To test the drivers, an integration project for the TQ Systems MBa8MPxL SBC must be created to include the compiled drivers from the custom Codeo pool.

Appendix B

Contents of the attached DVD

- `dwmac` ..... source codes of the dwmac driver
- `dwmac_ptp` ..... source codes of the dwmac PTP driver
- `test` ..... source codes of the testing scripts
- `thesis`
  - `src` ..... source form of the thesis in LaTeX format
  - `Satoplet-Jakub-MT-PTP.pdf` ..... text of the thesis in PDF format
- `README` ..... copy of this contents listing