CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Control Engineering

Bachelor's Thesis

NuttX RTOS Driver for Single Unshielded Twisted Pair Communication

Michal Matiáš

May 2025
Supervisor: Ing. Pavel Píša, Ph.D.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Matiáš**	Jméno:	**Michal**	Osobní číslo:	**492164**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Otevřená informatika**

Specializace: **Internet věcí**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Ovladač pro komunikaci systému RTOS NuttX po jednom sdíleném páru vodičů

Název bakalářské práce anglicky:

NuttX RTOS Driver for Single Unshielded Twisted Pair Communication

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Pavel Píša, Ph.D. katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **24.01.2025** Termín odevzdání bakalářské práce: **23.05.2025**

Platnost zadání bakalářské práce: **do konce letního semestru 2025/2026**

podpis vedoucí(ho) ústavu/katedry

podpis proděkana(ky) z pověření děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání

Matiáš Michal

Podpis studenta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Matiáš**	Jméno:	**Michal**	Osobní číslo:	**492164**
Fakulta/ústav:	**Fakulta elektrotechnická**				
Zadávající katedra/ústav:	**Katedra měření**				
Studijní program:	**Otevřená informatika**				
Specializace:	**Internet věcí**				

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Ovladač pro komunikaci systému RTOS NuttX po jednom sdíleném páru vodičů

Název bakalářské práce anglicky:

NuttX RTOS Driver for Single Unshielded Twisted Pair Communication

Pokyny pro vypracování:

The simplicity of the single shared pair communication, which allows interconnecting multiple units, is desired for many industrial applications with low and mid-data throughput requirements for its simplicity and cost. This is a typical area for CAN/CAN FD communication, which guarantees deterministic arbitration by priority for RT applications. However, mapping of typical TCP/IP-based communications to CAN FD frames adds additional complexity for less timing-demanding applications. The solution is CAN XL standard, but chips supporting CAN XL are still rare. Another option is the 802.3cg 10BASE−T1S variant of Ethernet communication. This project aims to test a combination of the latter option with NuttX RTOS and develop and integrate drivers for the NuttX project.
1) familiarize with NuttX RTOS operating system and perspective single pair automotive communication technologies
2) prepare short overview and analysis of available solutions; preliminary search reveals OnSemi 10BASE-T1S NCN26010 MAC with SPI interface to microcontroller as suitable choice
3) choose one of NuttX supported platforms as a base for chosen solution drivers development and implement driver
4) prepare some application for demonstration, i.e. simple DC motor control or more advanced PMSM motor controller with remote control over chosen communication standard
5) document the solution and demo application. Send the implemented driver for review to NuttX mainline and attempt to resolve received feedback.

Seznam doporučené literatury:

1. NuttX Apache project https://nuttx.apache.org/
2. OnSemi 10 Mb/s Industrial Ethernet MAC + PHY IC Controller NCN26010 (802.3cg 10BASE−T1S) Datasheet https://www.onsemi.com/products/interfaces/ethernet-controllers/NCN26010
3. Janoušek, J.: Open-source Motion Control on Mid-range and Small FPGAs; Master's Thesis; 2024; https://dspace.cvut.cz/bitstream/handle/10467/117046/F3-DP-2024-Janousek-Jakub-DP_Janousek.pdf
4. Espressif ESP32-C6 SoC, with a 32-bit RISC-V core https://www.espressif.com/en/products/socs/esp32-c6

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s):	Matiáš Michal
Personal number:	492164
Programme name:	Open Informatics

declare that I have elaborated the bachelor’s thesis entitled

NuttX RTOS Driver for Single Unshielded Twisted Pair Communication

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor’s and Continuing Master’s Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 21.05.2025

Michal Matiáš
....
student's signature

Acknowledgement / Declaration

Zde bych chtěl poděkovat vedoucímu práce Ing. Pavlovi Píšovi, Ph.D. za jeho cenné rady a diskuze nad technickými tematy.
Poděkování patří také mým rodičům, bez jejichž podpory bych nemohl studovat na vysoké škole.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 23. 5. 2025

....

Abstrakt / Abstract

Tato práce se věnuje evaluaci a integraci levného řešení pro multidrop komunikaci na krátkou vzdálenost s důrazem na spolehlivost a real-time vlastnosti, která lze použít ve vestavěných systémech. Na základě přehledu použitelných standardů je navrženo řešení sestávající z levných MCU zařízení v kombinaci s SPI MAC-PHY Ethernetovými zařízeními třídy 10BASE-T1S. RTOS Apache NuttX je navržen jako vhodný operační systém pro použitá MCU zařízení. Jedna kapitola je věnovaná přehledu vnitřní logiky síťových driverů NuttXu. Tato kapitola tvoří teoretický podklad pro hlavní výstup projektu, kterým je implementace konkrétního driveru pro SPI MAC-PHY NCV7410 od výrobce Onsemi. Ke konci je pozornost věnována použitým metodám pro evaluaci a testování implementovaného driveru. Poté jsou vlastnosti systému ověřeny pomocí jednoduchého drive-by-wire demonstrátoru.

Klíčová slova: NuttX, 10BASE-T1S, 10BASE-T1L, OA SPI Protocol, RTOS

Překlad titulu: RTOS NuttX driver pro komunikaci po jedné nestíněné dvojlince

The thesis aims to evaluate and integrate a low-cost solution for a small area multidrop communication with focus on reliability and real-time predictability usable in embedded systems. Based on a review of applicable standards, an integration of cheap MCU devices with 10BASE-T1S Ethernet SPI MAC-PHYs is proposed. The Apache NuttX RTOS is proposed as a suitable operating system for the MCU devices. An overview of NuttX network driver internals is provided in a dedicated chapter to provide background for the major outcome of the project – the implementation of Onsemi NCV7410 SPI MAC-PHY NuttX driver for the NuttX operating system. At the end, the methods used for an evaluation and testing of the implemented driver are discussed. The system properties are then verified on a simple drive-by-wire demonstrator.

Keywords: NuttX, 10BASE-T1S, 10BASE-T1L, OA SPI Protocol, RTOS

Contents /

/ Figures

Chapter 1
Introduction

The desire for digital communication in environments where focus is put on robustness, real-time properties, and the cost of the end devices, is common for various fields of industry. This led to the development of various communication technologies, serving various use cases. Some of them evolved into world-wide standards. This work proposes a communication solution integrating IEEE 10BASE-T1S Ethernet enabled SPI MAC-PHYs and the NuttX operating system.

Among the main advantages of the proposed solution are out-of-the-box compatibility with Ethernet standards and thus the TCP/IP protocol family, application code portability, open-source infrastructure, decent data rate, and a small form factor.

The second chapter lays out a set of requirements for the solution, then proceeds to provide an overview of selected communication standards. By the end of the chapter, 10BASE-T1S SPI MAC-PHYs and NuttX RTOS are selected as suitable technologies and are further discussed.

The chapter 3 presents the concepts behind NuttX network drivers. Different approaches to interfacing with the NuttX network stack are discussed. Descriptions of the IOB buffer structure and NuttX work queues are presented. The chapter is concluded by a note on a method common in NuttX for implementing inheritance in the C language.

The fourth chapter focuses on the implementation of the NuttX network driver for the NCV7410 SPI MAC-PHY. The emphasis is placed on the explanation of the internal logic of the driver.

In chapter 5, the setup for the evaluation of the system performance is presented. Results of the performance evaluation are shortly discussed. A few techniques used to test the system's features are mentioned.

The last chapter summarizes the work and discusses future goals.

Chapter 2
Technology Overview and Objectives

This chapter defines the target environment through a set of requirements for the communication system. Based on the requirements, a selection of communication standards is made and a brief overview of each selected standard is provided. At the end of the chapter, IEEE 10BASE-T1S is selected for the implementation of the communication system.

2.1 The Target Environment

To limit the selection of applicable technologies, the requirements for the designed system first need to be defined. For this work, the target environment is an environment where the following properties are needed.

- **Reliability**
- **Determinism**
- **Low Cost**
- **Multidrop Support**
- **Compact Size**
- **Ease of Integration**

Different subsets of the above properties are widely required in automotive, aerospace, industrial production, space applications, and other fields. A brief elaboration on the mentioned characteristics follows.

The reliability property ensures that a message sent between devices over the chosen channel arrives at the destination and arrives unaltered. The reliability is desired even in EMI noisy environments (e.g. tram yards, railway automation, high power servo systems). Due to this requirement, a wired solution is preferred.

Determinism refers to the predictability of the system. That is: a message sent over the channel arrives at the destination in a known bounded time.

Low-cost system in the context of this work is considered a system that allows individuals to use the system for prototyping or use in technological projects without the cost of the system being a major limiting factor. The ideal solution would allow low-cost and widely available MCUs to be used as end devices.

It is preferable that the system allows communication between multiple nodes in the network. By multiple nodes, support for up to at least three or more devices is expected. As in contrast with point-to-point only communication standards. This is expressed by the requirement for multidrop support.

The system should have a compact size to allow integration into embedded systems.

The ease of integration of the system is the property to fit well into a larger system (possibly comprising other currently used technologies) without excessive work. This demand is put on all layers of the target solution – physical, protocol, software.

2.2 Selected Communication Standards

This section presents a brief overview of a set of candidate communication standards that at least partly fulfill the declared requirements.

2.2.1 Local Interconnect Network (LIN)

Local Interconnect Network is a communication standard first defined for use as a low-cost, low-bandwidth – up to 20 kbit/s – communication solution in automobiles where the usage of CAN for all devices would be too costly. On the physical layer, a single conductor (and the ground reference) is used for the exchange of information. The conductor is shared by up to 16 devices. Among the devices, there is a single *commander* node and possibly multiple *responder* nodes. The commander initiates all data transactions on the bus. This means a collision cannot emerge if all nodes behave correctly. On the other hand, this also makes it impossible for other nodes to initiate communication. Also, this implies that the LIN bus is inherently half-duplex.

LIN bus intended workflow consists of the commander polling data from the individual responder nodes or sending configuration data to them. Therefore, while the LIN physical layer topology is of the type *bus*, the protocol logic rather imitates the *star* topology, where the commander is the central element and the responder nodes are addressed in a point-to-point manner.

As a native LIN physical layer is usually not present on common MCUs, a specialized physical layer device must be added to the design. The UART peripheral is typically used to interface with the physical layer device.

The LIN bus integrates well with the CAN bus. This property is often utilized in vehicles, where devices connected through the LIN bus create sub-networks to the backbone CAN network.

Using specialized hardware, it is possible to transfer LIN data over DC power lines using high-frequency modulation. Due to this, complexity and cost of cabling can be further reduced.

2.2.2 RS-485

The RS-485 is a physical layer specification. A balanced interface is used to achieve a good resistance against electromagnetic interference (EMI). A single twisted pair is used as a medium to carry the balanced signal. A full-duplex connection can be achieved with four conductors.

Multiple devices can be connected to the shared twisted pair using short stubs. The devices that are not transmitting at the moment can switch their output pins to the high-impedance mode to allow other nodes to transmit. Multidrop can be achieved through this capability. However, due to the fact that the RS-485 only defines the physical layer, the arbitration method must be implemented by the designer of the system, or another higher-level standard utilizing RS-485 must be used. Typical methods are central coordinator, token passing, or time division.

The maximum data rate depends on the length of the twisted pair segment. At 12 meters, the maximum data rate is 10 Mbit/s [1].

The RS-485 hardware is low-cost, widely used, and widely available. However, it should be noted that the price of the system increases when galvanic isolation is desired.

As RS-485 specifies only the physical layer, out-of-the-box compatibility with existing and used higher-level protocols cannot be achieved.

2.2.3 Controller Area Network (CAN)

Controller Area Network Protocol is a communication solution defined by a multitude of standards. Those standards define the data link layer, various options for the physical layer, and an application layer (CANopen). For its favorable properties, the CAN bus is the dominating serial network for embedded control systems in passenger cars [2]. The most significant properties of the CAN protocol are multidrop capability, error detection functions and automatic erroneous frame retransmission, and bus arbitration through data prioritization. The data rate of the classic CAN is typically 500 kbit/s. For a newer standard – CAN FD – the data rate can vary, ranging up to 5 - 8 Mbit/s.

The communication over the CAN protocol is organized into frames. Each frame consists of (among other fields) a message identifier and payload data. The payload data length is up to 8 bytes. In the case of the CAN FD, the payload data length can be up to 64 bytes. The identifier uniquely defines the type of the message that is being transmitted in the data payload. The identifier also defines the priority of the message. The identifier is at the start of each frame. Due to this clever design, collisions are resolved during the start of the frame transmission without incurring any delays needed for collision resolution. If multiple devices transmit data at the start of the frame transmission, the first one to see a logical zero on the bus while transmitting a logical one will stop transmitting until the next transmission opportunity. This way, the frame with the lowest ID is prioritized.

The CAN controller is available on many low-cost MCU chips. The physical layer driver must be provided separately. The physical layer drivers are available and low-cost. However, the cost increases if galvanic isolation is desired.

In the CAN ecosystem, several standardized application layer solutions are defined. These solutions provide a concise way for a programmer to be able to implement a desired communication between the nodes while being freed of the burden of dealing with hardware-specific details. Although these protocols are useful in the specific domain, the CAN protocol does not provide direct interoperability with communication stacks used elsewhere, namely TCP/IP protocol stack. This limitation stems from the fact that the two standards were not designed with mutual interoperability in mind. For example, minimum IPv4 MTU is 68 bytes. That is more than even the CAN FD can handle in one frame. This alone makes it difficult to bridge these two technologies.

2.2.4 CAN XL

CAN XL is historically the newest standard of the CAN family. While it retains backward compatibility with the classical CAN and the CAN FD standards, it offers up to 20 Mbit/s data rate and data payload sizes from 1 to 2048 bytes long [3]

In contrast to previous CAN standards, the CAN XL was designed with interoperability with internet protocols in mind. The large payload size allows even full Ethernet frame tunneling.

Although the promise of the CAN XL is appealing, the presence of the CAN XL controllers in consumer-grade MCUs has yet to emerge.

2.2.5 IEEE 10BASE-T1S

The 10BASE-T1S is an Ethernet standard defining 10 Mbit/s communication over a single unshielded twisted pair segment. The standard was designed for robust performance in electromagnetically noisy environments and with deterministic properties in mind [4]. The IEEE standard defines that half-duplex communication must be supported, and two mutually exclusive modes of operation are specified.

- Full-Duplex, Point-to-Point communication over up to at least 15 m segment
- Half-Duplex multidrop of up to at least 8 devices over up to at least 25 m mixing segment

Based on the requirements outlined in 2.1, the latter mode of operation seems more suitable for this work.

Due to the multidrop nature of the standard, the shared mixing element must be protected from transmission conflicts by employing an arbitration method. For this purpose, 10BASE-T1S uses the traditional Carrier Sense Multiple Access with Collision Detection (CSMA/CD).

The CSMA/CD allows all nodes to transmit at any time there is no traffic on the bus. This can lead to the situation where two or more nodes start to transmit data simultaneously. This is detected by the respective nodes. To resolve the conflict, each of the conflicting nodes sets its local timer to a random amount and waits until the timer elapses. The node whose timer elapses first starts its transmission. In the case where the collision is detected again, the described procedure repeats.

As a result of its design, the CSMA/CD can cause an unpredictable jitter in latency and degraded throughput. This is especially true for buses with heavy traffic, where most devices need to transmit data frequently. The 10BASE-T1S aims to satisfy the demand for predictability by introducing a special reconciliation sublayer called Physical Layer Collision Avoidance (PLCA) [4]. The PLCA aims to improve bus determinism and improve latency and throughput by distributing transmission opportunities to individual nodes in a round-robin manner. PLCA does not need to be set up in all nodes on the segment. Other devices may rely solely on the CSMA/CD and yet be able to communicate with the PLCA-utilizing devices. When using PLCA on all nodes on the mixing segment, the bus is behaving deterministically. If a node wants to transmit a packet, the time to wait for the transmit opportunity is upper-bounded by the maximum period of the PLCA transmission opportunity distribution cycle.

10BASE Ethernet MAC controllers are available in low-cost MCUs. Only the pairing with the appropriate 10BASE-T1S PHY must be done. In some cases, the native MAC controller is not available on the selected MCU, or the number of MCU pins needed by the respective PHY to communicate with the MAC would be too limiting. An alternative exists. A range of 10BASE-T1S MAC-PHY devices with the SPI interface is produced by several manufacturers. These devices share a common protocol for the SPI communication by OPEN Alliance [5]. The protocol specifies the mapping of SPI transfers on Ethernet frames and SPI transfers for configuration data exchange.

Galvanic isolation using cheap capacitors is provided in the base design.

2.3 Technology Choice Discussion

Based on the requirements for the solutions outlined in the section 2.1 and the selected standards properties, the decision was made to integrate the 10BASE-T1S standard using the mentioned SPI MAC-PHYs. The Onsemi NCV7410 MAC-PHY chip was chosen for the implementation and initial testing. The protocol used for the MAC-PHY SPI transfers is briefly described in 2.6.

The 10BASE-T1S in combination with the SPI MAC-PHYs was chosen due to a very good overlap with the outlined requirements.

Reliability is achieved through a robust physical layer design [4] and possibly enhanced by higher-level protocols from the TCP/IP stack.

Determinism can be achieved by utilizing the PLCA reconciliation scheme.

The MAC-PHYs are available at a moderate price, and using SPI as the interface to the host imposes minimal constraints on the hardware selection for the end device. A *low-cost* solution is therefore possible.

MAC-PHYs with *multidrop* capability will be used.

The MAC-PHYs are available in small FBGA packages, the end devices could be MCUs also featuring a small package size, and the space requirements for a single unshielded twisted pair cables are low. This allows the solution to be *compact* in size.

Finally, *ease of integration* is addressed by using an Ethernet standard. This allows standard TCP/IP networking. In many existing systems, Ethernet is used as a network backbone, and a specialized protocol such as CAN is used on the network edge, where embedded devices operate. By using 10BASE-T1S, the need for the specialized protocol disappears, and bridging of two mutually non-coherent standards is not needed. The *ease of integration* requirement is further addressed by OS selection (see 2.4).

2.4 Operating System Selection

Upon consideration, it was decided that the Apache NuttX operating system will be integrated. A solution without an operating system would be possible, but the use of an operating system has advantages.

One of the most important advantages of using an operating system is the application code portability. This goes hand-in-hand with the required ease of integration (2.1). If the end device hardware needs to be changed to a different chip or even architecture, possibly due to an upgrade or for other reasons, the required change to the application code should be minimal. Ideally, no change should be required. This is not possible without the use of an operating system.

Another advantage of using an OS is the possibility to use high-level features such as priority threading or the TCP/IP stack. These functionalities could, however, be provided by the respective SDK of the used end device hardware.

2.5 NuttX

NuttX is an open source real-time operating system with a small footprint that focuses strongly on providing strict standard compliance (POSIX) to support a rich multi-threaded development environment for deeply embedded processors [6].

Due to the strict adherence to the POSIX standard, NuttX theoretically stretches the application code portability to all POSIX-compliant targets.

NuttX is implemented in C.

2.5.1 NuttX Configuration

NuttX is a highly modular OS. Components of the OS can be included or excluded during configuration before NuttX is built. Almost all features of the NuttX operating system can be configured this way. Be it scheduling algorithms, device drivers, or support for particular networking protocols. The Kconfig language is used for this purpose. The process of configuring NuttX before the compilation is referred to as *build configuration*.

2.6 OPEN Alliance 10BASE-T1x MAC-PHY Serial Interface

The selected MAC-PHY uses SPI to interface the host MCU. The OPEN Alliance 10BASE-T1x MAC-PHY Serial Interface (OA) defines the standard capabilities of the MAC-PHY and the logic of the SPI data transfers between the host and the MAC-PHY [5].

On top of the standard four-wire SPI interface, the OA defines that one additional conductor is used to serve as an interrupt signal from the MAC-PHY to the host. As SPI transfers can only be initiated by the bus master, the interrupt signal is used to indicate that a special event has occurred, and that the host should use an SPI transfer to poll status data from the MAC-PHY.

For SPI transfers, the specification defines two subprotocols. One for exchanging configuration data, the other for the exchange of Ethernet packets. Both are briefly described in the following text.

2.6.1 Control Transaction Protocol

The Control Transaction Protocol (CTP) allows for configuration data exchange between the host MCU and the MAC-PHY.

The configuration is done by reading or writing MAC-PHY's internal registers. The OA specifies that the registers are addressed by a 4-bit Memory Map Selector (MMS) value and by a 16-bit offset (ADDR) into the memory bank selected by the MMS.

Each CTP transaction is initiated by the host sending a 32-bit header followed by variable length data. The header contains MMS, ADDR, write-not-read flag, length, and other fields. The first bit of the Control Transaction Protocol header distinguishes the CTP header from the Data Transaction Protocol header (see later).

When the data are being written to the MAC-PHY, the MAC-PHY should, upon receiving the whole header, echo the header back to the host and then continue echoing the data sent after the header. By this logic, a layer of verification is added.

If the data are being read from the MAC-PHY, the MAC-PHY should echo the received header the same way as when writing is done. After the echoed header, the requested register data is sent to the host. A special protection mechanism can be enabled, where each register is sent twice – the original register is followed by a ones' complement version of the original register.

The protocol length field in the header allows the exchange of up to 128 consecutive 32-bit registers in a single transaction.

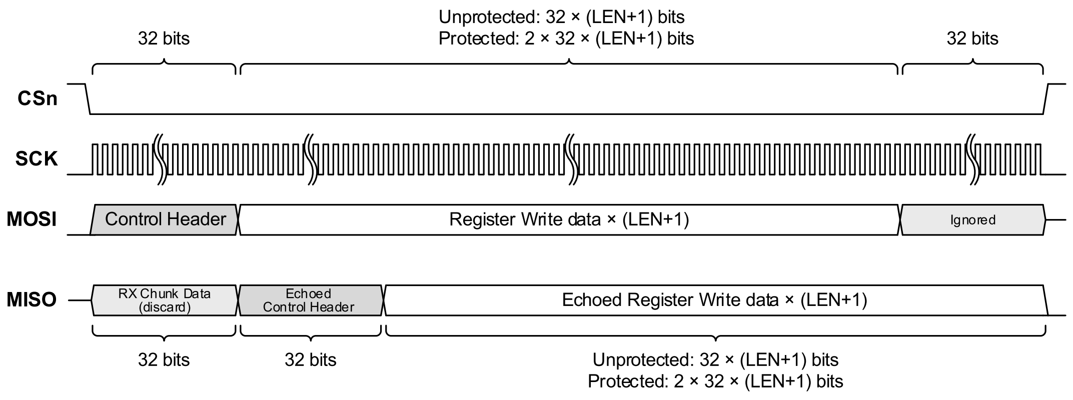
32 bits
Unprotected: 32 × (LEN+1) bits
Protected: 2 × 32 × (LEN+1) bits
32 bits
CSn
SCK
MOSI
Control Header
Register Write data × (LEN+1)
Ignored
MISO
RX Chunk Data (discard)
Echoed Control Header
Echoed Register Write data × (LEN+1)
32 bits
32 bits
Unprotected: 32 × (LEN+1) bits
Protected: 2 × 32 × (LEN+1) bits

Figure 2.1. SPI Control Transaction

2.6.2 Data Transaction Protocol

The Data Transaction Protocol (DTP) defines the mapping of Ethernet frames on SPI transfers.

The MCU hardware typically supports a limited number of bytes that can be exchanged during one SPI transfer. This number is typically much lower than the typical length of the standard Ethernet frame, let alone its maximum length of 1500 bytes. For this reason, the DTP introduces *chunks*.

The DTP chunk is a fixed-length array containing payload data and a 32-bit control field. The length of the payload data can be set during configuration. Depending on the specific MAC-PHY capabilities, lengths of 8, 16, 32, and 64 bytes are possible.

There are two types of chunks. The transmit chunk and the receive chunk. The transmit chunk is used to pass the data from the host to the MAC-PHY, while the receive chunk is used in the other direction. The 32-bit control field of the transmit chunk is at the start of the chunk and is called the *header*. The control field of the receive chunk is located at the last 32 bits of the chunk and is called the *footer*.

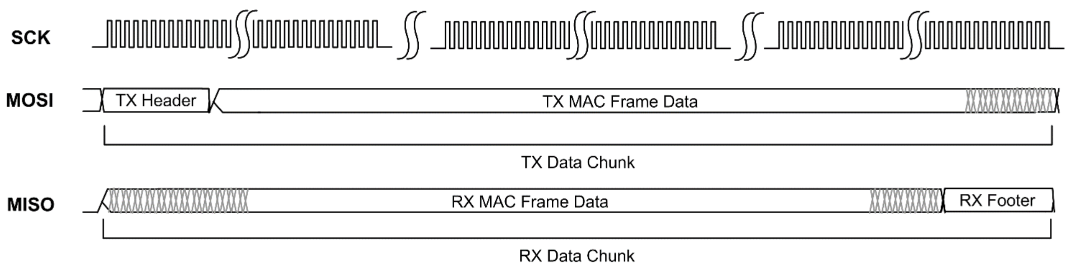

Figure 2.2. SPI Chunk Data Transaction

The header consists of fields informing the MAC-PHY whether the host is ready to receive data in a receive chunk and other fields.

The footer informs the host about the state of the MAC-PHY. The footer contains information about the minimal amount of available receive data chunks and the minimal number of transmit chunks that can be currently accepted by the MAC-PHY. Other fields are also present.

Both the header and footer contain information about whether there is valid transmit data in the chunk. If the chunk contains the start or the end of the frame, their respective positions are signaled in the respective control sequence. Both control sequences also contain the parity bit.

The following figures show the bit layout of the data chunk header and footer. A more detailed description is provided below. First for the TX header specific fields, then for the RX footer specific fields. Fields common to both the header and footer are explained in the last paragraph. For the full reference, see [5].

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DNC=1 1 bit	SEQ 1 bit	NORX 1 bit	RSVD 5 bits					VS 2 bits		DV 1 bit	SV 1 bit	SWO 4 bits			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RSVD 1 bit	EV 1 bit	EBO 6 bits						TSC 2 bits		RSVD 5 bits					P 1 bit

Figure 2.3. TX Data Chunk Header

The DNC (Data Not Control) bit distinguishes between the control transaction header and the data header.

Through the NORX (No Receive) bit the host indicates to the MAC-PHY that it will not receive RX chunk in the current transfer, therefore the MAC-PHY should retain the chunk in its memory.

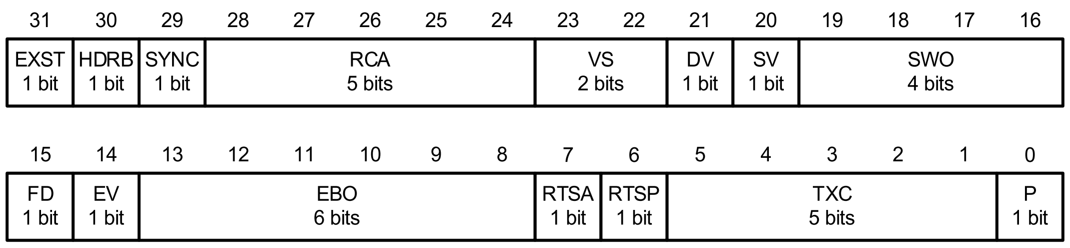

Figure 2.4. RX Data Chunk Footer

The EXST (Extended Status) bit indicates, that an event has happened that requires the host's attention and that cannot be contained in other fields of the header. This might be an error or other event that the programmer enabled during the configuration to raise this flag. Upon receiving, the host should use the control protocol to read appropriate status registers from the MAC-PHY to locate the source of this flag and possibly to perform a related action.

The HDRB (Header Bad) bit signals, that the MAC-PHY received a header with wrong parity.

The SYNC (Configuration Synchronized) bit value is tied to a bit in one of the configuration registers. This bit must be set by the host when the configuration of the MAC-PHY is complete. The importance of this flag is that it may happen that the MAC-PHY suffers reset during the operation. This flag gives the host the opportunity to detect this state and prceed accordingly.

The RCA (Receive Chunks Available) field in the footer indicates the minimal number of received chunks that are available to be read by the host. Similar to the RCA field is the TXC (Transmit Credits) field. It indicates the minimal number of transmit chunks that can be currently sent to the MAC-PHY without incurring the buffer overflow.

The FD (Frame Drop) signals to the host that an error occurred on the physical layer, and that the currently received Ethernet frame should be dropped by the host. (Note that this can only happen in the cut-through mode, where the frame is not stored fully in the MAC-PHY before sending to the host. Otherwise the affected frame is dropped in the MAC-PHY).

Both the header and the footer contain DV (Data Valid), SV (Start Valid), SWO (Start Word Offset), EV (End Valid), and EBO (End Byte Offset) fields. These fields carry the information about the Ethernet frame data possibly present in the rest of the chunk. More specifically these fields signal whether the frame data are present in the current chunk, whether there is a start or the end of the frame present in the chunk, and if so their respective offsets are indicated.

2.7 Testing MAC-PHY Hardware (Onsemi NCV7410)

For testing, the NCV7410 from Onsemi was selected. The NCV7410 is 10BASE–T1S Ethernet MAC-PHY compliant with the OA protocol supporting the multidrop mode of operation. Through the courtesy of the manufacturer, a few samples were provided for this thesis.

2.8 Testing Host Device Hardware (ESP32-C6)

For the initial testing the ESP32-C6 chip along with the ESP32-C6 DevkitC develpment kit was selected.

ESP32–C6 is a low–cost MCU manufactured by Espressif. The chip is driven by a 32-bit Risc-V primary core with up to 160 MHz clock speed and a secondary 32-bit Risc-V core that can be clocked up to 20 MHz. The main core features 512 kB of SRAM memory [7].

The chip features various functions and peripherals, for this thesis, mainly the support for the SPI bus and the NuttX support of the chip are important. During the implementation of the demonstrator device, the pulse counter peripheral was configured to implement decoding of a quadrature encoder.

Chapter 3
NuttX Network Drivers

This chapter introduces the reader to the general concepts in NuttX network drivers. Different approaches to interfacing the NuttX network stack are discussed. Two data structures are presented. IOB buffers – means of transporting data throughout NuttX, and work queues – a mechanism allowing code execution in an independent thread. The chapter is concluded by a note on implementation of inheritance in the C language.

3.1 A Network Driver

A network driver in an operating system is a piece of software that forms a functional layer between network hardware and the operating system network stack. Its main function is to accept incoming data from the networking hardware and pass it to the OS network stack in a form that the network stack expects, and vice versa – passing the data from the OS network stack to the networking hardware. Secondary functions are networking hardware configuration or statistics collection.

In NuttX, network driver usually runs in a dedicated thread, uses a work queue, or does a combination of both. The network driver is independent of the network stack in the context of invoking input-output operations. The network stack can turn the driver ON and OFF, and it can inform the driver about an available TX packet, but it is the sole responsibility of the driver to call the network stack code to acquire that TX packet and to pass it to the network device. The situation is the same in the RX direction. When the driver receives a packet from the network device, it must call the network stack code to pass the received packet to the network stack. A closer look at this interface is presented in the following section.

From the standpoint of the OSI model, the network driver lives in its data link layer.

3.2 NuttX Network Stack Interface

NuttX has an in-built network subsystem based on uIP project. The subsystem implements a standard TCP/IP stack and supports both IPv4 and IPv6 protocols. In order to allow connection between the network stack and network drivers, an interface is defined.

The interface consists of a network device driver state structure, callback functions, and other interfacing functions.

3.2.1 Network Device Driver State Structure

The network device driver state structure (`net_driver_s`) is a structure that defines a single network interface. It contains information used by the network stack and by the network driver. The fields closely related to the network driver are the following:

- `d_iob` and `d_buf` – these are two different pointers to the packet data buffer
- `d_len` – length of the packet in the `d_buf`
- `d_lltype` – type of the data link layer protocol, only Ethernet will be considered
- `ether` – structure containing a representation of the MAC address
- callback function pointers – these are described in the following section 3.2.2

The list above mentions two different pointers to the packet data buffer. Currently, the network stack and new network drivers internally use the IOB data structure for packet storage. As described in 3.3.2, this approach has several advantages. The pointer to the IOB buffer is stored in `d_iob`. The `d_buf` field is left for backward compatibility reasons. It points to a flat `uint8_t` data buffer, possibly the one inside the IOB buffer pointed to by `d_iob`. When the IOB buffer is not used, the `d_iob` field must be left `NULL`.

More information such as the IP address of the interface, interface name, configuration flag bits, and statistics is included in the network device driver state structure. These other fields won't be discussed as they are more relevant to the network stack than to the network driver.

3.2.2 Interfacing Callback Functions

The network driver callbacks are functions that should be defined by the network driver. These functions may then be called asynchronously by the network stack. There are six callback functions that should be implemented by the driver. The following is a list of the callback functions with a brief description for each.

- `d_ifup` – enable the networking hardware
- `d_ifdown` – shut down the networking hardware
- `d_txavail` – a TX packet is ready in the network stack, perform work to obtain it
- `d_addmac` – add the given MAC address to the hardware address filter
- `d_rmmac` – remove the given address from the hardware address filter
- `d_ioctl` – perform an IOCTL procedure

The behavior of the callbacks is not defined too strictly to accommodate the needs of various hardware devices. Depending on circumstances, implementation of some of the callbacks can be omitted. For example, some devices might not possess a MAC address filter; therefore, the implementation of `d_addmac` and `d_rmmac` would be pointless. Another example would be a design of the driver logic in which the driver periodically polls the network stack for TX packets. This means the `d_txavail` function implementation would be superfluous. In a situation when the implementation of one or more of the callback functions is not provided, the corresponding function pointers in the `net_driver_s` network device driver state structure must be set to `NULL` during initialization.

3.2.3 Network Stack Functions

The network stack provides the following functions to the network driver. The full signature of each function is provided:

- `int ipv4_input(struct net_driver_s *dev)`
- `int ipv6_input(struct net_driver_s *dev)`
- `void arp_input(struct net_driver_s *dev)`
- `int devif_poll(struct net_driver_s *dev, devif_poll_callback_t cllbk)`
- `void netdev_carrier_on(struct net_driver_s *dev)`
- `void netdev_carrier_off(struct net_driver_s *dev)`

The above list is not exhaustive; focus was put on functions relevant to the Ethernet data link layer protocol to align with the main topic of this thesis.

All the functions take a pointer to the `net_driver_s` network device driver state structure as an argument.

If the network driver receives an RX frame from the networking hardware, it is supposed to fill a buffer with the appropriate portion of the received frame (see 3.2.4). When the buffer is filled, `d_buf`, possibly `d_iob` (see 3.2.1) is set to point to the buffer. Depending on the L3 protocol enclosed in the received L2 frame, one of the above `_input` functions is called to pass the data to the network stack. In the case of Ethernet, the L3 protocol is determined using the 2 octets long Length/Type field at the end of the Ethernet header [4]. Upon returning, the `_input` function might leave an outbound packet in the `net_driver_s` structure. This is signaled by the `d_len` field being greater than zero. In that case, the network driver should process the packet in the `net_driver_s` and send it to the network.

When the network driver is ready to process a TX packet from the network stack (possibly after a call to the `d_txavail` callback), the driver should call the `devif_poll` function to poll the network stack for TX packets. Apart from a pointer to the `net_driver_s` structure, the `devif_poll` takes one more argument of type `devif_poll_callback_t`. This is a pointer to a callback function of the following definition:

- `typedef int (*devif_poll_callback_t)(struct net_driver_s *dev)`

The `devif_poll` traverses all active network connections, and for each of those connections, the callback is called. As the typedef above suggests, the `devif_poll` passes a pointer to the `net_driver_s` structure to the callback. The presence of a valid TX packet in the buffer pointed to by the `net_driver_s` buffer pointers is signaled by the `d_len` field. If the `d_len` field is greater than zero upon entering the callback function, it means a valid TX packet is present in the associated buffer and should be processed by the network driver. After the callback finishes its work, it should return zero if the driver is ready to process other TX packets. The `devif_poll` then continues traversing the active connections and calling the callback. In the case that the driver is not yet ready to process another TX packet at the end of the callback, the callback should return a non-zero value to end the polling. When the driver becomes ready again, a call to the `devif_poll` can be issued again.

It is worth noting that when the transmission operation to the network hardware device is not performed in the body of the `devif_poll`'s callback and is postponed to later, the `d_iob` buffer pointer should be moved to the driver and the `d_iob` should be set to `NULL`. Otherwise, the associated IOB buffer will be freed at the end of the `devif_poll` function, possibly causing data corruption. A legacy option is to provide a valid pointer to a flat buffer in the `d_buf` field by the driver code. Data is then copied from a network stack internal IOB buffer to the provided buffer in the body of the `devif_poll` function.

3.2.4 Expected format of Data Link Layer Packet

So far, the mechanism of the packet exchange between the network driver and the network stack has been described. The actual format of the packet passed between the network stack and the network driver must be mentioned as well.

Generally, the link layer packet format expected by the network stack depends on the type of the data link layer protocol that the given interface uses. This is determined by the `d_lltype` field in the `net_driver_s`. In the case of Ethernet, the packet consists of a 14 octet long Ethernet header and a variable length payload. The header consists of 6 octets of the destination MAC address, 6 octets of the source MAC address, and 2 octets of Length/Type field as defined in [4]. Some Ethernet frame headers might include a 4 octet tag field before the Length/Type field. This type of Ethernet packet is not supported, and if received from the networking hardware, it is the driver's responsibility to remove the tag from the frame before passing it to the network stack. The payload data follows after the header. Some networking hardware may include a frame check sequence (FCS) after the payload data in a received packet. Although a packet with an appended FCS will be processed correctly (the FCS would be perceived as padding) by the network stack, the FCS should be stripped off the packet by the driver, as a matter of good practice.

3.3 IOB Data Structure

Central to the NuttX network driver system is the IOB data structure. IOB stands for Input Output Buffer. As notable from its name, this structure encapsulates the data transferred between layers of the NuttX networking system. The data structure is described in the following text.

The IOB data structure is, in its core, a singly linked list. The structure consists of successive nodes, where each node contains a pointer to the next node. A single node is usually called an i/o buffer in the NuttX source code documentation. The entire data structure is commonly referred to as a buffer chain. I will try to follow this naming convention for consistency. Apart from the pointer to the next i/o buffer, an i/o buffer contains the actual buffer – an array of constant length for the data to be stored in. The length of the data buffer is given by the build configuration and is set to 196 bytes by default. A single i/o buffer further contains an offset value - how many bytes into the i/o buffer is the start of the data, and a length value – how many bytes of data are stored in the entry. All entries also contain the packet length value - this value, however, is valid only for the head of the buffer chain and tells how many bytes are stored in all i/o buffers of the buffer chain. In the NuttX network subsystem, one IOB buffer chain is used to store one packet. NuttX source code includes routines for managing data in IOB buffers, such as copying data from and into plain buffers. These routines are used in the driver implementation.

3.3.1 IOB Allocation Mechanism

During the system startup, a pool of free i/o buffers is allocated in a contiguous block of memory. The number of i/o buffers is determined by build setup options; in my setup, the number is 36. Initially, all the free nodes are linked together into one chain. When a NuttX subsystem needs an i/o buffer, an i/o buffer allocation routine is called from that subsystem. The i/o buffer is then taken from the free list head and passed as a pointer to the subsystem. When the subsystem no longer needs the i/o buffer, it calls the i/o buffer free function, and the i/o buffer is returned to the head of the free list.

3.3.2 Advantages of IOB buffers

Internet packets and other serializable i/o data usually come in a wide range of lengths. One option to accommodate chunks of data varying in size is to have a set of preallocated buffers of the same or larger size than the anticipated size of the longest of those chunks.

Each buffer is then assigned to a single chunk. The disadvantage of this approach, especially on memory-constrained systems, is the inefficient use of memory resources in a case where the system needs to deal with a large number of small chunks. In this scenario, the system could easily run out of the preallocated buffers, while most of the memory of the buffers would not be utilized.

The assumption behind IOB buffers is that the data chunk, which needs to be stored in the buffer, is usually smaller than its maximum possible size. It is therefore convenient to allocate a larger amount of small buffers that can be possibly linked together to make space for a chunk larger than a single small buffer. With the same amount of memory, this system provides much more flexibility than the naive solution described above. In case of dealing with small chunks, a larger number can be accommodated. When dealing with large chunks, a larger chunk can be accommodated.

Another advantage of IOB buffers is that it facilitates passing data between layers of the operating system. Before the IOB buffers were introduced to NuttX, data from network drivers was passed by a pointer to a flat data buffer. This buffer was usually statically allocated in the driver; therefore, when it was passed to the upper layers, the network driver must have waited until the buffer was processed (possibly copied) by the network stack before it could have been used again by the network driver. In the case of IOB buffers, the pointer to the buffer can be simply passed to the network stack (enclosed in the `net_driver_s` structure) and a new IOB buffer can be allocated immediately. The IOB buffer can then be processed and freed by the network stack independent of the driver.

3.4 Upper-Half Network Driver

In order to simplify interfacing with the network stack as previously described (3.2), another layer is present in NuttX. This layer lies between the network stack and the network device specific driver. In this setup, the part of the driver responsible for handling the device-specific logic is called the *lower-half driver*. The part responsible for interfacing with the network stack is called the *upper-half driver*. The upper-half driver takes care of IOB buffer management, statistics collection, and hides the intricacies of the `net_driver_s` structure from the programmer. As for the means of code execution of the upper-half driver code, the upper-half driver can be configured to either use the system-wide work queue or a dedicated thread. The driver implemented as part of this work relies on the upper-half network driver.

In a way, the interface between the lower and the upper drivers is very similar to the interface between a full driver and the network stack. Callback functions `ifup`, `ifdown`, `addmac`, `rmmac`, and `ioctl` are present in this interface as well, and the functionality is the same as before. The upper-half driver merely passes the respective calls from the network stack to the lower-half driver logic. Minimal intermediate action is performed.

What is different in the upper-half driver interface is the way the network packets are exchanged. The upper-half driver relies solely on IOB buffers for the data exchange. For packet transmission, the interface defines a simple callback.

- `int (*transmit)(struct netdev_lowerhalf_s *dev, netpkt_t *pkt)`

This callback function replaces the potentially complex polling as described in 3.2. The `netpkt_t` (netpacket) type in the second argument is used in the upper-half – lower-half logic to encapsulate one network packet. The `netpk_t` is summarized in 3.4.1. The `netdev_lowerhalf_s` is briefly discussed in 3.4.2.

When the callback is invoked, the lower-half driver has the opportunity to accept the packet and to take possession of it along with the responsibility of sending it to the network and freeing it later. The lower-half can also decline the packet by returning a negated error code such as `-EAGAIN`.

When a network packet is done transmitting to the network hardware, the lower-half driver may inform the upper-half driver about this event by calling the `netdev_lower_txdone` function. The upper-half driver is woken up and possibly passes new data to the lower-half.

When a network packet is received from the network hardware, the lower-half driver may inform the upper-half driver about this event by calling the `netdev_lower_rxready` function. The upper-half driver then invokes the following lower-half driver callback.

- `netpkt_t *(*receive)(struct netdev_lowerhalf_s *dev)`

Similarly to the `transmit` callback, but in the opposite regard, this callback should pass ownership of the received netpacket from the lower-half to the upper-half network driver. If no such packet is ready, `NULL` can be safely returned. Upon passing the received packet to the upper-half driver, the upper-half driver dispatches the packet to the respective `_input` function based on the registered network hardware device type (e.g. Ethernet, Bluetooth, CAN) and the packet header.

At the expense of lower flexibility, a more concise interface is provided. Utilization of the upper-half network driver is currently the community-preferred approach to implementing network drivers for NuttX.

3.4.1 Netpackets

Netpackets are used to exchange network packets between the upper-half and the lower-half drivers. They are represented by the `netpkt_t` data type. The `netpkt_t` type is identical to the `iob_s` IOB buffer structure (see 3.3). A thin layer of abstraction above IOBs has been built as part of the upper-half – lower-half logic. The purpose of the `netpkt_t` is to clearly distinguish namespaces. Also, if it is later decided that netpackets should use a different underlying data structure than the `iob_s`, no lower-half driver honoring the interface needs to be changed provided that the netpacket interface remains the same.

The following is a selection of functions from the netpacket interface.

-
  ```
  netpkt_t *netpkt_alloc(struct netdev_lowerhalf_s *dev,
                         enum netpkt_type_e type)
  ```

-
  ```
  void netpkt_free(struct netdev_lowerhalf_s *dev, netpkt_t *pkt,
                   enum netpkt_type_e type)
  ```

-
  ```
  int netpkt_copyout(struct netdev_lowerhalf_s *dev, uint8_t *dest,
                     const netpkt_t *pkt, unsigned int len, int offset)
  ```

-
  ```
  int netpkt_copyin(struct netdev_lowerhalf_s *dev, netpkt_t *pkt,
                    const uint8_t *src, unsigned int len, int offset)
  ```

-
  ```
  unsigned int netpkt_getdatalen(struct netdev_lowerhalf_s *dev,
                                 netpkt_t *pkt)
  ```

The `netpkt_alloc` allocates a buffer from the list of free buffers if possible. If the allocation is successful, respective quota are decreased (3.4.2). The `netpkt_type_e` enum distinguishes between the TX and the RX buffer.

The `netpkt_free` frees a previously allocated buffer. Quota are increased.

The `netpkt_copyout` is used to copy `len` bytes of data at `offset` from the netpacket into a flat buffer.

The `netpkt_copyin` is used to copy `len` bytes of data from a flat buffer into the netpacket at `offset`.

The `netpkt_getdatalen` returns the number of bytes stored inside the buffer.

The `netpkt_setdatalen` sets the length of data inside the buffer. This doesn't need to be used if the `netpkt_copyin` function is used to fill that netpacket with data, but the function can also serve to strip excess data from the end of the netpacket.

The figure 3.1 presents a high-level view of layers of the NuttX network system along with buffer data structures for the data exchange between the layers. The SPI interface is added as it is relevant to the driver implemented in this work.

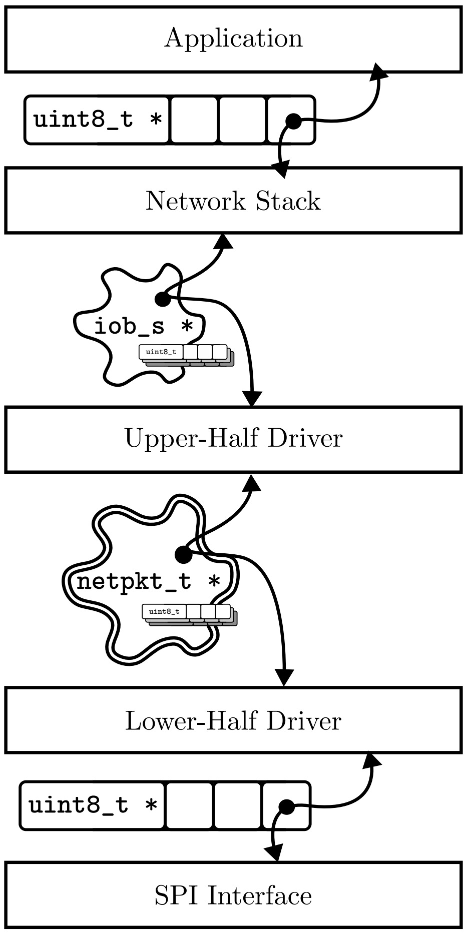

Figure 3.1. NuttX Network System Layers

3.4.2 Lower-Half specific data

The lower-half specific data are stored in the `netdev_lowerhalf_s` structure. This structure is used to define a single lower-half driver. It contains very little additional information on top of the `net_driver_s` structure. Apart from the `net_driver_s` structure, it encapsulates callback functions described above and netpacket quota – predefined maximum numbers of TX and RX netpacket buffers that can be held by the driver. By the mechanism described in the section about inheritance (3.6), the `netdev_lowerhalf_s` can be extended to contain data specific to the particular driver.

3.5 Work Queue

A work queue is a mechanism that, in its basic form, allows asynchronously executing blocks of code in a first-come, first-served manner. In NuttX, work queue is often utilized in network drivers. The main advantage of a work queue is its simplicity of use in comparison with setting up a dedicated thread.

3.5.1 Executing Code in NuttX Work Queues

A work scheduled to be executed in the work queue is defined by its state structure. Before first use, this structure needs to be initialized to all zeros, but after that, it is completely managed by the work queue logic. When a work needs to be scheduled into a work queue, a call to the `work_queue` function must be issued. This function takes an id of a work queue, a pointer to the work state structure, a pointer to a callback function to be executed, and a minimum delay to wait before execution. The work queue id selects one of the work queues present in the system.

3.5.2 Work Queue Drawbacks

Work queues must be used with caution, as there is a range of pitfalls to be aware of.

One such pitfall is that the work queue interface can be used by unrelated parts of the operating system. This might lead to deadlocks [8]. Let's say one part of the system holds an IOB buffer and the code to free the buffer is executed through the work queue. The second part of the system needs to allocate a buffer and uses a blocking IOB allocation function in a body of its work queue worker. If there are no other free buffers available, the work queue is in a state of deadlock. The buffer cannot be released by the first part of the system because it is waiting for the worker of the second part of the system to finish. The second worker cannot finish because it is waiting for the IOB buffer.

This problem could be eliminated by configuring NuttX to use multiple threads to run the work queue (`CONFIG_SCHED_LPNTHREADS`). The problem with deadlocks is solved, but other problems may arise from the fact that the jobs scheduled in the work queue are now not guaranteed to run sequentially but can be executed in parallel.

Another solution might be to create a dedicated work queue solely for the network driver.

Both of these solutions, of course, increase the memory footprint of the OS.

Only short callback functions should be scheduled in the work queue; otherwise, jitter may be increased in the system.

As a conclusion, if not done properly, work queues usage might cause the system's real-time properties to deteriorate or even result in a deadlock.

3.6 Inheritance in C

Drivers in NuttX often employ the upper-half – lower-half paradigm. The upper-half driver typically implements logic that is common to a certain class of devices (e.g. pressure sensors, network devices). The lower-half then provides support for a single device type of the class (e.g. BMP180, NCV7410). As described in section 3.4, the network drivers are no different.

In programming, in order to provide a common interface to a higher-level logic, while allowing flexibility by extension at the lower level, inheritance is commonly used.

Even though the C language does not feature explicit support for inheritance, [1] the principle of inheritance can be achieved by using the following technique. The concept is shown schematically in an example from NuttX.

First, the parent struct is defined.

```
struct netdev_lowerhalf_s
{
  const struct netdev_ops_s *ops; /* reference to the callbacks */
  ...
  struct net_driver_s netdev; /* interface to the net stack */
};
```

When the parent is defined, the child structure can be defined as a structure with *the first field* being the instance of the parent structure. Arbitrary other fields can (but do not need to) be defined.

```
struct ncv7410_driver_s
{
  struct netdev_lowerhalf_s dev;

  struct spi_dev_s *spi; /* SPI interface */
  ...
};
```

When the reference to the child structure needs to be passed to the parent logic, which is not aware of the child's type, the parent structure field is simply passed. As described in 4.3, in NuttX, this is done during the driver registration.

```
netdev_lower_register(&priv->dev, ...);
```

In the above snippet, the `priv` argument is a pointer to an instance of the `ncv7410_driver_s` structure, allocated and initialized beforehand by the child's logic.

When the parent invokes one of the interfacing callbacks to pass execution to the child's logic, the reference provided before by the child (registration above) is passed as an argument. This argument is then simply retyped back to the child's structure type.

```
static int ncv7410_ifup(struct netdev_lowerhalf_s *dev)
{
  struct ncv7410_driver_s *priv = (struct ncv7410_driver_s *) dev;
  ...
```

This mechanism is allowed due to the fact that the C language guarantees that a pointer to a structure object points to its initial member, and vice versa [9].

As a note aside, the parent structure from the above example could be considered an *abstract type*, as the child must implement the interfacing functions in order to be used in any meaningful way. By the definition provided in [10], the abstract type cannot be instantiated. In C, the `netdev_lowerhalf_s` can be of course instantiated, but the meaning of such instantiation with no supporting logic would be questionable.

[1] NuttX enforces the C89 definition of C

Chapter 4
NCV7410 Network Driver Implementation

This chapter aims to describe the implementation steps of the NuttX network driver for the NCV7410 MAC-PHY.

To be able to use the driver on a host MCU, support must be provided on two levels. First, a general driver must be implemented. This driver is hardware-agnostic. When such driver is implemented, supporting hardware-specific code must be provided in the so-called *board support package* (BSP) in order to support the specific host hardware used. This low-level code initializes needed peripherals on the host board/chip level. In the case of this work, those peripherals are the SPI bus and the GPIO interrupt. The following text is focused mainly on the host MCU-agnostic part of the code. The section 4.3.1 is dedicated to the hardware-specific initialization.

The upper-half – lower-half driver paradigm was used (see 3.4) to implement the driver. The implemented driver is therefore a lower-half driver. Depending on context, notation *driver* will be used to denote the lower-half driver or a full-driver as a working unit comprising the upper-half and the lower-half driver.

4.1 Driver Specific Data

The inheritance mechanism described in 3.6 is used to define driver-specific data on top of the data in the `netdev_lowerhalf_s` structure. The structure `ncv7410_driver_s` is created.

The `ncv7410_driver_s` contains the following additional fields.

A mutex for protecting the driver from data races (more in 4.8)

- `mutex_t mutex`

A pointer to the hardware-agnostic structure to operate the SPI periphery (4.3.1)

- `struct spi_dev_s *spi`

Two `work_s` instances for the work queue interface:

- `struct work_s interrupt_work` – interrupt handling (4.7)
- `struct work_s io_work` – data exchange (4.6)

Current state of operation – can be one of `NCV_RESET`, `NCV_INIT_DOWN`, `NCV_INIT_UP`

- `uint8_t ifstate`

Two fields indicating the current state of NCV7410's data buffers

- `int txc` – number of chunks that can be currently written to the MAC-PHY
- `int rca` – number of chunks that are available to be read from the MAC-PHY

Pointers to TX and RX IOBs along with four accompanying fields (described in 4.6)

- `netpkt_t *tx_pkt`
- `netpkt_t *rx_pkt`
- `int tx_pkt_idx`
- `int rx_pkt_idx`
- `int tx_pkt_len`
- `bool rx_pkt_ready`

4.2 Configuration Exchange Primitives

It was mentioned before that the OA defines a protocol for exchanging configuration data between the host MCU and the MAC-PHY (see 2.6.1). Apart from this, the OA defines memory regions present in the MAC-PHY device. Each such region is identified by its MMS number. When reading or writing a MAC-PHY register, the MMS and the address (ADDR) of the register in the region are placed in the control header. These two fields uniquely identify a single register in the MAC-PHY.

To provide a configuration interface to the MAC-PHY registers for the driver logic, the three following functions are defined.

-
  ```
  static int ncv_write_reg(struct ncv7410_driver_s *priv,
                           oa_regid_t regid, uint32_t word)
  ```

-
  ```
  static int ncv_read_reg(struct ncv7410_driver_s *priv,
                          oa_regid_t regid, uint32_t *word)
  ```

-
  ```
  static int ncv_set_clear_bits(struct ncv7410_driver_s *priv,
                                oa_regid_t regid,
                                uint32_t setbits, uint32_t clearbits)
  ```

All three functions modify a single register on the MAC-PHY. The first two functions write/read into/from the register from/into the `word` argument. The third function performs a read-modify-write operation on behalf of the caller. The register is first read, its contents are then OR'd with the `setbits` argument and AND'd with the one's complement of the `clearbits` argument. The resulting value is then written back to the MAC-PHY.

The destination/source register is identified by the `regid` argument of the type `oa_regid_t`. This type is defined in the accompanying header file of the driver. Under the hood, this type is a 32-bit number that encapsulates MMS and ADDR values for a particular register. Macros for `oa_regid_t` construction from MMS and ADDR as well as macros for the MMS and ADDR extraction from the `oa_regid_t` type are defined.

The OA specifies that some MMS-defined memory regions are common among all OA MAC-PHYs, while other regions are vendor-specific. With this, and plans for future logic generalization in mind, the macros with address declarations were defined. For every MAC-PHY register that needs to be accessed by the driver logic, a set of macros in the accompanying header file is defined. Every such register has a defined macro for its MMS, ADDR and REGID (of the `oa_regid_t` type). If any subfields or flags of the register are used, then there is a macro defining the respective position and the mask of the subfield or flag. The general format for the naming of the macros is `OA_<regname>_<suffix>` for the register residing in the OA-generic memory regions, where the `<suffix>` is either `MMS`, `ADDR`, `REGID` or the name of a subfield or flag. The format `<prefix>_<regname>_<suffix>` is used for the vendor-specific registers. At this stage, the `<prefix>` is always `NCV` as there are no other OA devices supported other than the NCV7410. The `<suffix>` is used the same way as for the OA-generic registers. The OA generic macros were named to reflect naming in the OA specification [5]. The naming for the NCV7410-specific macros is taken from the NCV7410 datasheet [11].

4.3 Driver Initialization

Before the driver can be used, it first needs to be initialized and registered with the operating system. Whether the driver will be initialized or not is configured in the build configuration. The following sections describe the driver initialization in chronological order.

4.3.1 Board-Specific Initialization

The MCU hardware first needs to be initialized in order to be able to interface with the MAC-PHY. The driver initialization therefore begins during the board initialization. During the boot of NuttX the `board_late_initialize` function is called by the OS. This function passes execution to the board-specific logic implemented in the BSP. The flow of execution is dispatched to the `esp_bringup` function (in the case of ESP32-C6 board). In this function, a call to start the NCV7410 driver is invoked. This is done by calling the `board_ncv7410_initialize` function.

In the `board_ncv7410_initialize` function, the SPI interface and the GPIO pin used for the interrupt signal are initialized. A pointer to the SPI interface along with the interrupt pin IRQ number are passed to the general driver logic in a call to the `ncv7410_initialize` function. As there is presently no universal mechanism for the general driver logic to control a GPIO pin, the hardware interrupt of the interrupt pin must be enabled by the board-specific logic *after* the `ncv7410_initialize` returns.

4.3.2 General Driver Initialization

In the `ncv7410_initialize` function, the memory for the `ncv7410_driver_s` is allocated by a call to the `kmm_zalloc` function. The `spi` field is initialized using the pointer to the SPI interface passed from the board-specific logic. The driver is now able to use the configuration exchange primitives (4.2) to configure the MAC-PHY. A reset to the MAC-PHY is issued. If successful, the driver status `ifstate` field is set to `NCV_RESET`. The factory-assigned MAC address is read from the MAC-PHY and copied into the respective field in the `net_driver_s` structure residing inside of the `netdev_lowerhalf_s dev` field. The `ncv_interrupt` (see 4.7) function is attached to the IRQ number passed from the board-specific logic. The mutex is initialized. The packet quotas are set to the predefined values. The lower-half function callback pointers are initialized by the pointers to the respective functions. The `netdev_lower_register` function is called. By this, the operating system is informed about the driver and can take needed steps.

4.3.3 Configuring the MAC-PHY

The MAC-PHY is configured by the first call to the `ncv7410_ifup` function. This usually happens right after boot. A configuration very similar to the one found in the datasheet [11] is used. The setup of the OA protocol and the MAC address filtering are mentioned in the following sections. After the successful configuration, the `ifup` field in the driver specific structure is set to `NCV_INIT_UP` and is ready to exchange data packets with the network.

4.3.4 OA Protocol Setup

The OA protocol is set up during the MAC-PHY configuration. The setting is the following. If the RX chunk contains a start of a frame, then the start of the frame is always placed at the start of the chunk. This could be disabled to allow for more efficient transfers, but the driver logic would need to be modified. The RX cut-through mode is enabled. In this mode, the MAC-PHY does not collect the whole received Ethernet packet before sending to the host. Rather it sends the chunks to the host as the data come. This leads to a slight decrease in latency. On the other hand, the driver logic needs to be enhanced by logic to drop the packet when the Frame Drop (FD) flag is found in the data chunk footer (see 2.6.2). Lastly, default chunk payload size is set to 64 bytes. This makes the total length of the data chunk 68 bytes long.

Note that the cut-through mode is disabled in the TX direction as the host cannot guarantee to feed data to the MAC-PHY fast enough. During the transmission of packets longer than a couple of chunks, the TX buffer underflow error happens on the MAC-PHY.

4.3.5 MAC Address Filtering

By default, the MAC-PHY passes all the Ethernet packets received on the common segment regardless of the target MAC address field. On segments with more than two hosts, this would decrease the performance of the hosts by having to deal with packets that are not addressed to them. To remove this problem, a MAC address filter is present in the MAC-PHY's hardware. This filter needs to be configured. The NCV7410 features four MAC address filter slots. Each slot consists of an address field and a mask field. To the address field, the address that should be allowed to pass through the filter is filled. The mask field defines which bit positions in the address field must match, and which bit positions are not checked by the filter.

In normal operation, a single filter slot is used to store the factory-assigned MAC address of the device mentioned in the previous steps. Note that during the build configuration, the promiscuous mode can be selected in which the filtering is disabled. The host then can be used, for example, for network eavesdropping.

4.4 Switching the Interface On and Off

The driver interface defines the `ifup` and `ifdown` functions. These functions can be called by the network stack at any time to cause the interface to shut down if currently up, or to turn on if currently down. In the NuttX shell, the user can call the `ifup` or the `ifdown` commands to invoke the function of the corresponding name.

The implementation in the NCV7410 driver logic of the functions is following. The `ifdown` function first cancels all driver specific tasks that may be scheduled in the work queue. This is done by calling the `work_cancel` function twice. Once with a pointer to the `interrupt_work`, once with a pointer to the `io_work` – both fields of the `ncv7410_driver_s`. After this, the MAC-PHY is disabled. This is done by modifying the respective MAC-PHY registers through the Control Transaction Protocol (2.6.1) using the `ncv_set_clear_bits` utility function (4.2). After this step, the state of the driver buffers is reset to the state after the driver first initialization. All the ongoing transactions are dropped by this. As the last step, the `ifstate` field in the `ncv7410_driver_s` is set to `NCV_INIT_DOWN`. The `ifdown` function then returns.

The `ifup` function is similar to the `ifdown` function. Due to the fact that the `ifdown` is responsible for resetting the driver buffers, the `ifup` function is freed from this task. On the other hand, the `ifup` function checks the `ifstate` field and if `NCV_RESET` is found, the `ifup` function is obligated to call the MAC-PHY configuration routines. If those routines finish successfully, the `ifstate` is set to `NCV_INIT_DOWN` state.

4.5 Asynchronous Code Execution

Some device drivers are implemented in a way that all routines are executed synchronously in the body of the data-gathering callback function. Others need to communicate with the device hardware more frequently than the callback functions are being called – some of their code, therefore, needs to run asynchronously, independent of the rest of the system. The case of network drivers lies predominantly in the latter category. Several options for the asynchronous execution exist.

One option would be to accommodate all asynchronous work into the *interrupt service routine* (ISR) assigned to the MAC-PHY interrupt signal. This approach is highly problematic due to several reasons. The ISR would need to handle SPI transfers. However, the SPI interface as currently provided by NuttX does not support invocation from the ISR context. Other hardware-related problems may arise. Also, this approach doesn't account for transmitting data. Data transmission executed in the `transmit` callback would therefore block the upper-half driver at least for the duration needed to transmit the entire packet being transmitted. This might negatively affect performance.

Other option would be to run all the asynchronous code inside a dedicated kernel thread. This option allows the most flexibility. Additionally, a priority value can be set for a kernel thread. On the other hand, kernel thread introduces increased memory footprint and a notable implementation overhead.

Simpler than using a dedicated thread, while retaining the demanded core functionality of asynchronous code execution, is using the system-wide *work_queue* (see 3.5) for running the asynchronous code. This approach was used in this work.

4.6 Buffer Management and Data Exchange

In order to be able to pass data between the upper-half driver and the network device, the driver needs some intermediary storage. This is provided by using netpackets (3.4.1).

The pointers to netpackets are stored in the `tx_pkt` and the `rx_pkt` fields of the `ncv7410_driver_s` structure.

The way the buffers are used to exchange data and how they are allocated is described in the following text.

4.6.1 Transmitting Data

When the TX packet is accepted from the upper-half, the `tx_pkt_idx` is set to zero, and the `tx_pkt_len` is set to the value returned by the `netpkt_getdatalen` function. Subsequently, the data is fed to the MAC-PHY using the SPI Data Transaction Protocol (2.6.2). The `netpkt_copyout` function is used to copy the data to a flat buffer for the SPI transfer in a chunk-by-chunk manner. The `tx_pkt_idx` is used as the offset argument to the `netpkt_copyout` function, increasing by the chunk payload size with each chunk sent.

When the `tx_pkt_idx` reaches the value of `tx_pkt_len`, the packet is considered to be sent, the netpacket is freed, and the `netdev_lower_txdone` function is called to inform the upper-half.

4.6.2 Receiving Data

If the receive chunk footer signals the presence of the start of a frame inside the chunk, the `rx_pkt_idx` is set to zero. Similarly to the data transmission, the data from the subsequent chunks with valid receive data are then copied into the RX netpacket using the `netpkt_copyin` function with the `rx_pkt_idx` as an argument.

When the receive chunk footer signals the end of the frame, the frame data remaining in the last chunk is copied to the netpacket. The last 4 bytes containing the FCS are then stripped from the netpacket using the `netpkt_setdatalen` function. Before calling the `netdev_lower_rxready` function, the `rx_pkt_ready` flag is set. This flag indicates to the rest of the logic of the driver that this netpacket is not available for writing.

After calling the `netdev_lower_rxready`, the RX netpacket must wait inside the lower-half driver until the `receive` callback is called. When that happens, the `rx_pkt` is set to `NULL` (after saving to a temporary variable), the `rx_pkt_ready` is set to `false`, and the netpacket is passed (from the temporary variable) to the upper-half. The upper-half driver is responsible for freeing the RX packet.

4.6.3 I/O Work

The logic described above is solely (apart from the callback functions) executed by a single function in the work queue. The name of the worker function is `ncv_io_work` and it encapsulates all the subroutines needed for the data exchange according to the OA protocol. The worker presents itself to the work queue interface using the `io_work work_s` instance in the lower-half driver structure.

To limit the time spent in the work queue (see 3.5.2), the `ncv_io_work` function always exchanges exactly one chunk over the SPI.

First the chunk exchange is prepared. The header is set accordingly to the current state of buffers, and transmit data is possibly copied from the TX netpacket to the TX buffer for the SPI exchange.

After the preparation step, the SPI transfer is performed. If the transmission of a full packet is finished, the according steps are taken (4.6.1).

The incoming buffer is decoded, RX data are possibly copied to the RX netpacket.

As the last step, the `ncv_io_work` function checks whether another TX/RX transfer is possible. If so, the next run of the `ncv_io_work` is scheduled using the work queue.

4.6.4 Netpacket Allocation

In the case of TX packets, the netpacket allocation does not need to be addressed by the lower-half driver at all. The upper-half driver is completely responsible for the TX packet allocation.

In the case of RX packets, the netpacket allocation must be done by the lower-half driver. In each run of the data exchange function, the `ncv_can_rx` is called. This function is used to tell the calling function whether it should set the No Receive (NORX) flag in the chunk header or not. As part of its simple logic, the RX packet allocation is addressed. The function first checks if there is data to be read, i.e. whether the `rca` field in the lower-half structure is greater than zero. If not, the RX netpacket does not need to be allocated. Then the function checks the `rx_pkt_ready` flag. If the `rx_pkt_ready` is `true`, the netpacket cannot be touched, and the function returns `false`. Continuing, if there is a netpacket already allocated, the function returns `true`. Finally, if no packet is allocated, the function tries to allocate the RX packet. The function then returns `true` if the allocation was successful. Otherwise, `false` is returned.

4.6.5 Quota Considerations

Quotas of the network driver are a concept that is part of the upper-half – lower-half network driver logic. It represents numbers of TX and RX netpackets, that can be allocated simultaneously by the full-driver. In accordance with the driver logic and due to the following considerations, the quota number for the TX netpackets is set to 1. The quota number for the RX netpackets is set to 2 (the second netpacket can be allocated after the currently received netpacket is passed to the upper-half).

Due to the fact that the NCV7410 MAC-PHY's TX and RX buffers both feature 4 kB of space, it was decided that it is enough for the driver to operate only one TX and one RX buffer. It is relied upon the MAC-PHY buffers to accommodate the incoming data during the time when the packet is in the `rx_pkt_ready` state. This is between the frame reception being signaled to the upper-half driver by calling the `netdev_lower_rxready` and the upper-half accepting the data using the `receive` callback.

The previous is, in fact, only important in the RX direction. If it is later found that the mentioned premise is unreliable, the `rx_pkt` pointer will be replaced by a fixed-length statically allocated array of `netpk_t` pointers. The same will be done with `rx_pkt_idx` and `rx_pkt_ready`. An index to the new arrays denoting the active packet will be allocated, and the active packet selection logic will be implemented.

Possibly a better workaround would be using the IOB queue – an IOB buffer FIFO queue present in NuttX. However, this feature is not yet integrated with the upper-half's `neptk_t`. Even though the `netpkt_t` type is identical to the `iob_s` structure, the abstraction built by the upper-half logic would be ideologically disturbed.

4.7 Interrupt Handling

As mentioned before, an interrupt signal may be sent from the MAC-PHY to the host.

Due to steps taken during the initialization, the assertion of the interrupt signal is dispatched to the `ncv_interrupt` ISR function. The `ncv_interrupt` is designed to be as short as possible. Therefore, it only schedules execution of the `ncv_interrupt_work`. The `ncv_interrupt_work` might be called an interrupt task in some operating systems, but the `ncv_interrupt_work` is executed using the work queue. The notation *interrupt worker* might be therefore preferable. The interrupt worker is represented to the work queue interface by the `interrupt_work` field in the lower-half driver structure.

As suggested in [5], the interrupt worker first polls the receive chunk footer from the MAC-PHY. This is done by setting the No Receive (NORX) flag and clearing the Data Valid (DV) flag in the transmit chunk header.

At this point the Extended Status (EXST) flag should be checked and the possible source of its assertion should be determined. However, at this stage, this is not implemented.

The `txc` and `rxa` fields are updated based on the values in the corresponding fields of the footer. As the last step, the interrupt worker checks if the data can be exchanged with the MAC-PHY according to the updated `txc` and `rxa` and the state of netpacket buffers. If so, the interrupt worker schedules execution of the `ncv_io_work` function using the work queue.

4.8 Thread Synchronization

During normal operation, up to four threads may be trying to access the network driver specific structure. To prevent problems caused by different threads modifying the structure in an unpredictable way, atomicity (from the thread perspective) of parts of the code is enforced by using the *mutex* lock.

NuttX work queues can be configured to use multiple threads for the execution of the scheduled tasks during the build configuration. The sequential execution is therefore not guaranteed by the work queue in the general case. Without this option, the number of threads concurrently accessing the driver structure would be lower than the number mentioned above, but still greater than one.

For illustration, the mentioned four threads would be the interrupt worker, io worker – both invoked by the multi-threaded work queue, the upper-half calling `receive` or `transmit` and the network stack calling `ifdown`, `addmac` or `rmmac`.

In order to protect the data from race conditions, relevant parts of code in functions `ncv_interrupt_work`, `ncv_io_work`, `transmit`, `receive`, and `ifdown` were wrapped inside the `nxmutex_lock` – `nxmutex_unlock` pair.

Additionally, upon locking the mutex, the locking thread always checks the `ifstate` value for the correct state in the current context. This prevents invalid operations, such as the io worker trying to perform data exchange after the MAC-PHY has been shut down.

The wrapped code in the `ncv_interrupt_work`, `ncv_io_work`, and the `ifdown` functions includes SPI transfers. This might evoke a feeling of decreased performance. However, it is shown that the performance penalty incurred is minimal. All the functions in the set of wrapped functions may be blocked by an SPI transfer. When any of the functions with an SPI transfer is blocking another such function, there is no way of improving performance as the access to the SPI interface is inherently exclusive. When any function is blocked by the `ifdown` function, the performance is not a concern at all, as the interface is shutting down. When the `transmit` or `receive` functions are blocked, no performance penalty is applied. These functions only modify the driver-specific struct to make a request for data exchange that, in the end, needs to be processed by the `ncv_io_work` function.

Only concern might be the blocking of the work queue by an SPI transfer as it may increase the latency of the work queue. However, this cannot be mitigated by better synchronization. If the multi-threaded work queue configuration is used along with DMA SPI transfers, the work queue can wake up another thread and continue execution there while the SPI transfer is in progress. Note that during the DMA SPI transfer, the execution is blocked by waiting on a semaphore. The semaphore gets released after the transfer is done.

Chapter 5
System Evaluation and Testing

This chapter presents methods that were used for the evaluation and testing of the implemented system. To provide a quantitative estimate of the system latency and throughput, `ping` and `iperf` utilities were used, respectively. Other techniques for testing are mentioned. The process of evaluation and testing was facilitated by the use of special hardware – the EVBUM2876, a 10BASE-T1S equipped USB-C external network interface compatible with a regular PC. For verification of the properties of the implemented system in a real-world application, a simple drive-by-wire demonstrator was created.

As a result of testing under load and in various configurations, several bugs were found in the code and fixed.

5.1 Performance Evaluation

This section describes how the performance of the system introduced in past chapters was evaluated. Focus is mainly put on the data throughput and latency of the system.

5.1.1 Network Utilities

For measuring the system qualities, the `iperf` and `ping` utilities were used.

To measure the throughput of the system, the `iperf` utility was used. The `iperf` utility is based on the server-client model. One host poses as the server in the model, the other as the client. When the client connects to the server, data is sent in one direction from the client to the server. The throughput of the data is measured. The `iperf` utility offers two basic modes of operation – the UDP mode and the TCP mode. According to the protocol selected, the exchanged data is passed using one of the selected protocols.

For latency measurements, the `ping` utility was used. The `ping` utility works on top of the ICMP protocol. For the ICMP `echo request` message addressed to a certain host in the network, the ICMP `echo reply` message is expected back from the host. Apart from information about reachability, the round-trip time (RTT) statistics can be acquired by the `ping` utility. The `ping` utility has also been used in order to introduce artificial traffic into the network. The `-f` (flood) option of the `ping` utility was used for this purpose.

5.1.2 Testing Network

For the evaluation of the system, a small network comprising three host devices has been assembled. Two devices are the ESP32-C6 hosts. The third end device is a PC. The schematic representation of the testing network is in the figure 5.1. As seen in the figure, all interfaces feature 2 connectors. Note that both connectors of each device are directly electrically connected.

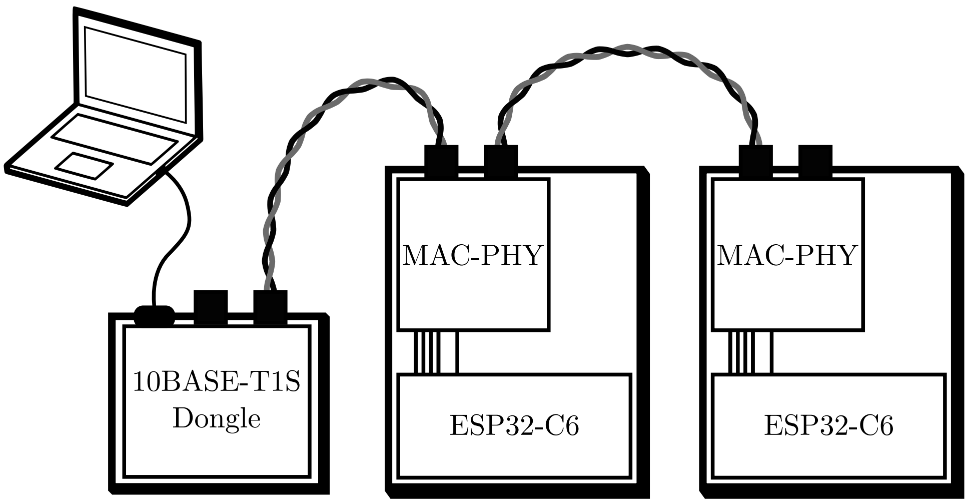

Figure 5.1. Schematic Representation of the Testing Network

5.1.3 Evaluation Scenarios and Results

The following scenarios were tested. The verbatim output of the utilities can be found in the appendix A.

- The `ping` (ICMP, 56 bytes of payload) response latency between the PC host and one of the ESP32-C6 hosts.
  - Average latency of 0.797 ms was measured, 3.784 ms maximum.
- TCP and UDP data throughput between two ESP32-C6 hosts and no additional traffic.
  - 3.06 Mbits/sec reached by the TCP, 5.09 Mbits/sec by the UDP.
- TCP and UDP data throughput between two ESP32-C6 hosts with added artificial traffic by the PC host flooding one of the ESP32-C6 hosts with `ping`, the `ping` latency was also measured.
  - 2.90 Mbits/sec reached by the TCP, 4.67 Mbits/sec by the UDP.
  - For TCP, the PC host measured the average RTT of 1.294 ms, 5.094 ms maximum.
  - For UDP, the PC host measured the average RTT of 1.830 ms, 4.349 ms maximum.

From the measured data, one clear observation can be made. When more traffic is introduced to the network, the latency rises, while the throughput declines. Setting of the PLCA in the network could lessen the negative impact of the added traffic. As mentioned in 6, implementation of this feature is planned.

Note that the data throughput and latency can be influenced by various aspects. The measured numbers are provided more as guidance to make an approximate assessment about the system properties and to provide a comparison between cases with and without artificial traffic, rather than presenting them as a definitive benchmark.

Various values for the size of packets were tested by the `ping` utility. Setting the size of the packet to an arbitrary size in the range defined for the standard Ethernet frame did not result in losing any packets.

5.2 Testing

This section briefly mentions other techniques used for testing.

In order to verify the proper function of the MAC address filter setting in the MAC-PHY, the `arping` tool was used. This tool is able to send ARP and IP pings to the specified host. During testing, it was used to send an IP ping similarly to the `ping` utility. In `arping`, however, the data link layer (MAC) address of the target host can be modified. The option `-t` is used for this purpose. Arpings were sent with the MAC-PHY's MAC address and with some other address. Then it was observed whether the requests with the correct address are passed to the MCU, and if the wrong requests are filtered by the MAC-PHY. Note that a similar test can be done without needing a special utility in a network with three or more nodes. However, at this stage, only two interfaces were available to us.

The driver code was carefully analyzed during the testing. A problematic time window was located, where, due to a bug in thread synchronization, a certain thread race condition could theoretically cause the driver to crash. The time window was too short for the probable vulnerability manifestation. The time window was therefore artificially inflated by adding a busy delay implemented as a for-cycle. The presence of the vulnerability was verified, and the bug was fixed.

The Wireshark software was used in various stages of implementation and testing.

5.3 Demonstrator Device

As an example of a real-world application, a simple drive-by-wire system was created. The system consists of two hosts. Both of the hosts are equipped with a small DC motor with a quadrature encoder sensor. The quadrature encoder sensor is supported by NuttX. Reading the position is therefore simplified to a mere call to an `ioctl` function on an open file descriptor of the registered quadrature encoder sensor driver. The motors are controlled using an H-bridge module via the combination of GPIO and PWM signals. The setting of the GPIO and PWM is done through the same mechanism as the quadrature encoder position is being read.

From the two devices in the network, one device is the *controller*, the second device is the *controlled*. The controller's motor is used only as the source of the position, therefore it does not need to be driven. The controller periodically reads the position of the motor shaft and sends it as a 32-bit number to the controlled over a TCP connection. The controlled receives the reference position and moves accordingly. For the control of the DC motor, the controlled implements a simple P regulator.

For the operation of the controller, a single thread is enough. The controlled uses two threads. One thread is responsible for the TCP transfers, the other implements the function of the P controller. A mutex is used to protect the common structure for the storage of the reference value.

Chapter 6
Conclusion and Further Work

In the extent of this thesis, a low-cost solution for a wired multidrop communication has been implemented. The emphasis was placed on reliability, predictability, and ease of integration. The solution integrates 10BASE-T1S SPI MAC-PHYs with host MCUs running the NuttX operating system. Data throughput rates of up to 5 Mbits/sec were measured with maximum RTT latencies below 10 ms. The throughput and worst-case latency properties could be further improved by setting up the PLCA reconciliation in the network.

Due to the selection of the NuttX RTOS as an operating system for the end devices, and as a result of utilizing the Ethernet standard in the core of the system, the application code is highly portable.

The work implemented in this work will be submitted to NuttX mainline for review and integration. The pull request with the changes is in preparation.

The work on the project will continue in collaboration with the NuttX community. Funding through the Google Summer of Code 2025 program has been granted for the project extension. As part of the extension, the logic defined by the OA will be separated from the NCV7410 specific logic to allow code reuse in NuttX drivers for other OA SPI MAC-PHYs. Support for PLCA configuration will be introduced to NuttX. Possibly through the definition of new IOCTL calls and integration with network utilities present in NuttX. Support for other host platforms will be provided. Currently, support for boards featuring the Microchip SAMV7 ARM processor is planned.

During the work on the demonstrator device, several obstacles were met in the form of bugs present in the code for the ESP32-C6 quadrature encoder support. With support from the NuttX community, most of the problems were resolved. In the process, source of one of the bugs was identified and the proposed solution has been merged into the NuttX mainline.

After due discussion with the NuttX community, parts of this thesis could be transformed into NuttX documentation.

Appendix A
Output of Network Evaluation Utilities

- The response latency between the PC host and one of the ESP32-C6 hosts

```
root@michal-ThinkPad-X250:~# ping -f 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

--- 10.0.0.2 ping statistics ---
17795 packets transmitted, 17794 received, 0,00561956% packet loss
rtt min/avg/max/mdev = 0.724/0.797/3.784/0.051 ms
```

- TCP and UDP data throughput between two ESP32-C6 hosts with no artificial traffic

```
nsh> iperf -c 10.0.0.2
     IP: 10.0.0.4
mode=tcp-client sip=10.0.0.4:5001,dip=10.0.0.2:5001, interval=30, time=30

           Interval         Transfer         Bandwidth
   0.00-  30.10 sec   11501568 Bytes    3.06 Mbits/sec
```

```
nsh> iperf -u -c 10.0.0.2
     IP: 10.0.0.4
mode=udp-client sip=10.0.0.4:5001,dip=10.0.0.2:5001, interval=30, time=30

           Interval         Transfer         Bandwidth
   0.00-  30.10 sec   19169856 Bytes    5.09 Mbits/sec
```

- TCP and UDP data throughput between two ESP32-C6 hosts with added artificial traffic by the PC host flooding one of the ESP32-C6 hosts with `ping`, the `ping` latency is also provided.

```
nsh> iperf -c 10.0.0.2
     IP: 10.0.0.4
mode=tcp-client sip=10.0.0.4:5001,dip=10.0.0.2:5001, interval=30, time=30

           Interval         Transfer         Bandwidth
   0.00-  30.10 sec   10895360 Bytes    2.90 Mbits/sec
```

```
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.

--- 10.0.0.4 ping statistics ---
30049 packets transmitted, 30048 received, 0,0033279% packet loss
rtt min/avg/max/mdev = 0.726/1.294/5.094/0.694 ms
```

```
nsh> iperf -u -c 10.0.0.2
     IP: 10.0.0.4
mode=udp-client sip=10.0.0.4:5001,dip=10.0.0.2:5001, interval=30, time=30

           Interval         Transfer         Bandwidth
   0.00-  30.10 sec   17580096 Bytes    4.67 Mbits/sec
```

```
root@michal-ThinkPad-X250:~# ping -f 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.

--- 10.0.0.4 ping statistics ---
18704 packets transmitted, 18703 received, 0,00534645% packet loss
rtt min/avg/max/mdev = 0.728/1.830/4.349/0.783 ms
```

References

[1] Kugelstadt, Thomas. Application Report: The RS-485 Design Guide. *Texas Instruments*. 2021. Available from `https://www.ti.com/lit/an/slla272d/slla272d.pdf`.

[2] CAN in Automation. *Controller Area Network classic (CAN CC)*. Available from `https://www.can-cia.org/can-knowledge/can-cc`.

[3] CAN in Automation. *CAN XL (extended data-field length)*. Available from `https://www.can-cia.org/can-knowledge/can-xl`.

[4] IEEE-802.3. IEEE Standard for Ethernet. *IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018)*. 2022. Available from DOI 10.1109/IEEESTD.2022.9844436.

[5] OPEN Alliance TC6. OPEN Alliance 10BASE-T1x MAC-PHY Serial Interface. dec, 2021. Available from `https://opensig.org/wp-content/uploads/2023/12/OPEN_Alliance_10BASET1x_MAC-PHY_Serial_Interface_V1.1.pdf`.

[6] *Appache NuttX*. Available from `https://nuttx.apache.org/`.

[7] Espressif. *ESP32-C6 Series Datasheet*. Rev. 1.2.

[8] Ashton, Brennan. *Work Queue Deadlocks*. Available from `https://nuttx.apache.org/docs/latest/components/net/wqueuedeadlocks.html`.

[9] ANSI/ISO. American National Standard for Programming Languages - C. 1992.

[10] Wikipedia. *Abstract Type*. Available from `https://en.wikipedia.org/wiki/Abstract_type`.

[11] Onsemi. *Automotive Ethernet Transceiver (MAC-PHY) 10BASE-T1S MultiDrop NCV7410*. Rev. 0.

Appendix B
List of Abbreviations

AND	Bit AND Operation
BSP	Board Support Package
DMA	Direct Memory Access
EMI	Electromagnetic Interference
FIFO	First In First Out
ISR	Interrupt Service Routine
MAC	Media Access Controller
MAC-PHY	Device integrating both the PHY and the MAC
MCU	MicroController Unit
OR	Bit OR Operation
OS	Operating System
PHY	Physical Layer Device
PLCA	Physical Layer Collision Avoidance
RTOS	Real Time Operating System
RTT	Round-Trip Time
RX	Receive
SDK	Software Development Kit
SPI	Serial Peripheral Interface
TX	Transmit
XOR	Exlusive OR logical operation