TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture

Chair of Distributed and Networked Systems

Master's Thesis

Design and Implementation of an NFC Library for RIOT OS

Nico Behrens

Born on: 31st July 2000 in Wismar
Matriculation number: 5145955

2nd December 2025

First referee
Prof. Dr. Matthias Wählisch

Second referee
Prof. Dr. Florian Tschorsch

Supervisor
M.Sc. Mikolai Gütschow

Statement of authorship

I hereby certify that I have authored this document entitled *Design and Implementation of an NFC Library for RIOT OS* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 2nd December 2025

Nico Behrens

Abstract

This thesis presents the design and implementation of a comprehensive NFC library for RIOT OS, addressing the critical gap in NFC support for constrained IoT devices. The work introduces the `nfcdev` interface, a novel hardware abstraction layer that bridges the heterogeneity of NFC devices while preserving access to device-specific features. The `nfcdev` interface strikes a balance between software flexibility and hardware capabilities, making it suitable for diverse NFC chipsets ranging from simple RF frontends to sophisticated NFC controllers.

The library provides comprehensive support for multiple NFC technologies (NFC-A, B, F, V) and implements high-level functionality for common use cases: reading and writing NFC tags (T2T, T4T, MIFARE Classic), tag emulation, and peer-to-peer communication. Four device drivers were implemented (nRF52840, PN532, PN7160, ST25R3916B) to validate the interface's flexibility and demonstrate its practical applicability across different hardware platforms.

Evaluation results show that the library achieves competitive performance compared to established NFC libraries while being specifically optimized for resource-constrained environments. Memory footprint, execution times, and energy consumption remain within acceptable bounds for real-world IoT deployments. The library rivals existing solutions in timing benchmarks and provides a more comprehensive feature set tailored for embedded systems, filling a significant gap in the RIOT OS ecosystem and enabling developers to integrate full-featured NFC capabilities into IoT applications without requiring deep expertise in NFC protocols or hardware-specific implementations.

Contents

List of Acronyms

1 Introduction

The chapter lays out the motivation and the ultimate goal of this thesis.

1.1 Motivation

Near Field Communication (NFC) has become ubiquitous in modern life. From contactless payment systems and digital ticketing to access control [21]. NFC enables seamless, short-range wireless communication in countless everyday scenarios.

While NFC has found widespread adoption in smartphones and other high-end consumer devices, its potential in the Internet of Things (IoT) remains untapped. Constrained embedded devices could benefit significantly from NFC capabilities.

RIOT OS, an open-source operating system specifically designed for the IoT, currently provides only limited NFC support. The NFC implementation fails to leverage the full potential of NFC and leaves users and developers without a unified, comprehensive solution for NFC-enabled IoT applications. Existing NFC libraries primarily target user-space applications on regular computers, or they come with limitations to the supported NFC protocols and a cumbersome interface.

Furthermore, the diversity of NFC hardware presents a significant challenge: different NFC chipsets offer vastly different capabilities and programming interfaces. Some provide high-level protocol handling in firmware, while others expose only low-level access. This heterogeneity makes it difficult to develop portable applications that can support various NFC devices.

1.2 Goal

This thesis aims to design and implement a comprehensive, modular NFC library for RIOT OS that addresses the limitations of existing solutions and enables full-featured NFC support on constrained IoT devices.

The primary objective is to create a unified hardware abstraction layer that bridges multiple NFC devices with varying capabilities while preserving access to device-specific features. This interface must be flexible enough to accommodate devices ranging from simple RF frontends to sophisticated NFC chips, yet simple enough to facilitate straightforward driver development and application programming.

Building upon this foundation, the library should provide high-level support for common NFC use cases: reading and writing NFC tags — including proprietary consumer-market tags — in their entirety but also by directly accessing embedded NFC Data Exchange Format (NDEF) messages, emulating tags to provide data to external NFC devices, and enabling P2P communication. To validate the design, the implementation will include four NFC device drivers that simultaneously demonstrate the interface's ability to accommodate diverse hardware. Each of these features and drivers must be implemented with careful attention to resource constraints, ensuring that memory usage, execution times, and energy usage remain within the bounds of typical embedded systems and other NFC libraries.

The evaluation will assess memory footprint, execution times, and energy consumption, comparing results against existing NFC libraries to establish the library's suitability for real-world IoT deployments and proving its usefulness in the RIOT ecosystem.

2 Background

This chapter will introduce RIOT OS as the target OS for the NFC library. It will then go into great detail about the broader NFC standard and its protocols. The chapter ends with an overview of the NFC hardware that will be considered for further design and implementation decisions.

2.1 RIOT OS

RIOT OS is an open source operating system for the Internet of Things (IoT) [11]. It has a small memory footprint, and it can target many CPU architectures. Among them are ARM7, ARM Cortex M, AVR, RISC-V, and others [35]. Although intended for embedded devices, it can also be compiled for native to make it run on x86 and ARM32 as a user space application for Linux. RIOT OS supports hundreds of boards and features drivers to make use of the board's peripherals [34, 36]. RIOT has a UART, SPI, and I2C driver to communicate with peripherals.

RIOT supports network protocols like Bluetooth Low Energy, LoRaWAN, and 802.15.4. It only has limited support for NFC in the form of two drivers for the PN532 and the nRF52840. The high-level NFC libraries are limited to Type 2 Tag (T2T) emulation and an NFC Data Exchange Format (NDEF) library.

2.2 Near Field Communication

Near Field Communication (NFC) is a short-range, wireless communication technology that enables two compatible devices to exchange data when they are brought close together. NFC emerged from previous Radio-Frequency Identification (RFID) standards. RFID introduced non-powered *tags* that would react to a powered *reader's* RF field. The connection could then be used to exchange data.

Various ISO/IEC RFID standards were unified by the NFC Forum and given new names and a streamlined nomenclature. The NFC Forum specifications split NFC into *NFC technologies* that can be found in the NFC Forum Digital and Activity specifications [20, 24]. The four NFC technologies are *NFC-A* and *NFC-B*, which are both based on the ISO/IEC 14443 standard [13,

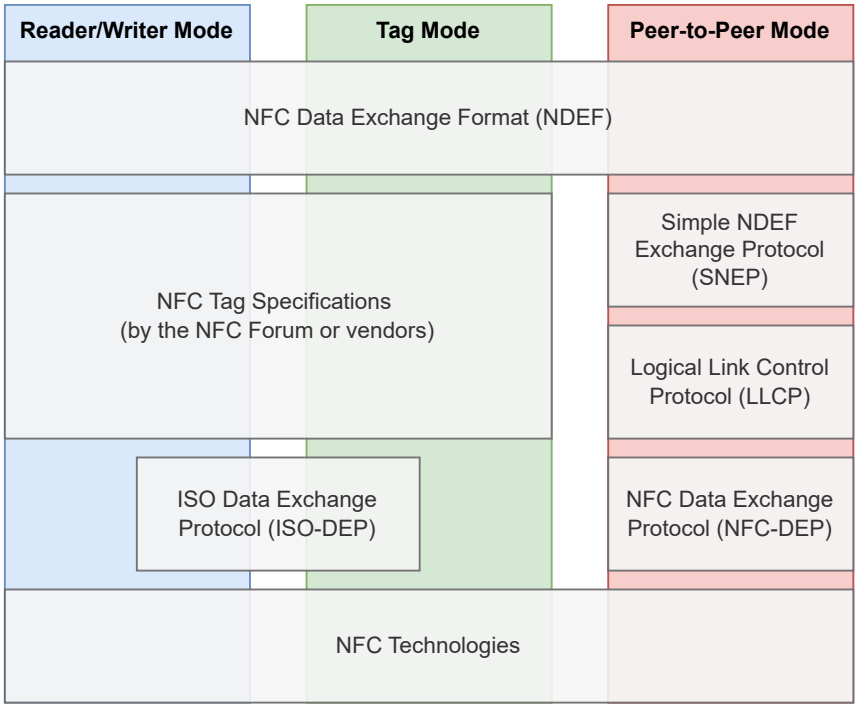

Figure 2.1: The three operation modes require different protocols. The reader/writer mode only works when used with another NFC device in tag mode. The ISO-DEP protocol is only used for type 4 tags.

14, 18], *NFC-F*, which is based on the JIS X6319 standard [10], and *NFC-V*, which is based on the ISO/IEC 15693 standard [16, 17, 27].

NFC operates at a frequency of $13.56\,\mathrm{MHz}$. The operation modes include the reader/writer mode and the tag mode, which were already part of RFID. A tag can also be emulated by an NFC device; this is why the tag mode is sometimes called *tag emulation* when a device acts as a tag. NFC adds a third mode to allow for both devices to initiate communication, enabling Peer-to-Peer (P2P) communication as defined in the ISO/IEC 18092 standard [23]. The different operation modes can be seen in Figure 2.1.

The NFC Forum defines NFC tag types 1, 2, 3, 4, and 5. Vendors define their own NFC tags, like the NXP MIFARE Classic, MIFARE Ultralight, or MIFARE DESFire tags. Only type 4 tags and tags based on type 4 tags use the ISO Data Exchange Protocol (ISO-DEP). P2P communication is built on top of the Simple NDEF Exchange Protocol (SNEP), the Logical Link Control Protocol (LLCP), and the NFC Data Exchange Protocol (NFC-DEP). In 2020, the NFC Forum introduced the Tag NDEF Exchange Protocol (TNEP), which is built on top of the tag specifications. NFC uses its own data format called the NFC Data Exchange Format (NDEF) that is used in all modes to encode application data.

Throughout the different standards, specifications, and manuals, there are often different terms used for the same concepts. NFC is always built around a *poller* starting the communication with a *listener*. For the case of tag interaction, this poller is a *reader/writer* (R/W), and the listener is a *tag*. The ISO/IEC 14443 standard [13–15, 18] uses the terms Proximity Coupling Device (PCD) for the poller and Proximity Integrated Circuit Card (PICC) for the listener. Some documents and protocols use initiator and target for the same roles.

2.3 Communication Modes

The NFC Forum defines an *active* and *passive* mode of communication. During passive communication, an active device (the *poller*) builds up an RF field, and the passive device (the *listener*) modulates the already existing field to send information to the active device.

Passive communication is the more common mode of communication and is crucial for tag communication because tags do not require power unless they are emulated. This communication mode sets NFC apart from other standards like Bluetooth or WiFi, where both devices are required to create an RF field. Passive communication can also be used for P2P communication.

With active communication, both devices build up their own RF field to send data to the other device. RF fields are created in an alternating fashion to prevent collision. This communication is exclusively used in NFC-DEP P2P communication.

2.4 NFC Technologies

The various NFC technologies emerged from different industry standards and have limited compatibility with each other. The NFC Forum is the organization that has labeled and streamlined these standards. Most information on NFC technologies provided by the NFC Forum can also be found in specifications issued by other organizations that will be cited in the respective technology's section. These specifications use a different nomenclature that will not be used.

Each NFC technology is split up into a digital protocol [20] part, which describes the modulation and framing of data, and an activity [24] part, which describes the flow chart for individual activities. All technologies have four activities:

1. an anti-collision activity to determine all listeners,
2. a device activation activity to activate a specific high-level protocol on the listener,
3. a data exchange activity that determines data exchange,
4. and an optional device deactivation activity that is not used for all protocols.

Out of these four, the anti-collision activity is the most complicated — requiring most of the commands defined by each technology. The device activation is usually limited to a single command exchange. The data exchange can be summarized as a repeated sending and receiving of high-level protocol data in standard frames. Device deactivation is handled according to the high-level protocol or when the RF field is removed.

2.4.1 NFC-A

NFC-A is the most common NFC technology. It is used for NFC Forum tags type 1, 2, and 4, as well as proprietary consumer-market tags by MIFARE. A real-world application would be the German *Personalausweis* eID [3]. NFC-A is based on ISO/IEC 14443 Type A [13, 14, 18].

Digital Protocol

Transmission Speeds NFC-A only supports $106\,\mathrm{kbit/s}$ for NFC-A in passive communication. In active communication, i.e. the active mode of P2P communication, NFC-A can be used with $212\,\mathrm{kbit/s}$, $424\,\mathrm{kbit/s}$ and $848\,\mathrm{kbit/s}$. The higher data rates are not supported by the hardware covered in the later chapters.

Frame Format NFC-A uses Short Frames, Standard Frames, and Bit Oriented Standard Frames. The Short Frames and Bit Oriented Standard Frames are only used in the anti-collision activity.

A Short Frame is made up of a Start of Frame (SoF), 7 data bits, and an End of Frame (EoF). This frame type is only used during the anti-collision activity.

Standard Frames are used for data exchange and consist of a SoF, a variable number of data bytes with an additional odd parity bit, and an EoF. Only Standard Frames carry a 2-byte CRC in the End of Data (EoD).

Bit Oriented Standard Frames are used for the anti-collision activity and are made up of two parts resulting from a Standard Frame. Part 1 is always sent by the poller, and part 2 is always sent by the listener in response to part 1. A few other constraints must be met:

- the sum of data bits in both parts is 56
- the data bits in part 1 must be in the range 16 to 48
- therefore, the data bits in part 2 must be in the range 8 to 40
- if the split occurs after the 8th data bit of a byte, then the related parity bit is added after the last bit of part 1
- if the split occurs at another position within a byte, then no parity bit is added after the last data bit of part 1, and the first parity bit of part 2 is undefined

Commands There are a handful of commands available for NFC-A. The table consists of the commands and their respective responses. The ISO/IEC 14443 standard uses a different terminology for some commands. This different terminology is used in older manuals for NFC-A capable NFC chips, like in the manual for the PN532 [2]. Therefore, their ISO/IEC counterpart is noted here, too. The mapping is taken from the NFC-A chapter of the NFC Forum Digital Specification [20].

The exact structure of most commands is omitted because they are not required to understand the anti-collision loop, but the SEL_RES response is shown in 2.2 as it contains information on the listener's capabilities at the end of the anti-collision activity, indicating higher-level protocol support. The 2-byte SENS_RES is used to determine the length of the listener's ID (NFCID1), and it contains platform information that can be used to identify the vendor of the tag.

Timing To understand later design decisions, it is crucial to outline the timing constraints for an NFC-capable device. It is especially important to look at the maximum timings to not miss a deadline when sending a frame. A violation of a deadline is treated as a timeout error, and the frame is ultimately discarded. Only the timings for the common $106\,\mathrm{kbit/s}$ bit rate are considered.

Command	Response	Checksum	Frame Type	Purpose	ISO alias
ALL_REQ		No	Short	Probing for listeners	WUPA
SENS_REQ		No	Short	Probing for listeners	REQA
	SENS_RES	Yes	Standard	Response to probe	ATQA
SDD_REQ		No	Bit Oriented	Anti-collison loop	ANTICOLLISION
	SDD_RES	No	Bit Oriented	Anti-collison loop	ANTICOLLISION
SEL_REQ		Yes	Standard	Listener selection	SELECT
	SEL_RES	Yes	Standard	ACK to selection	SAK
SLP_REQ		Yes	Standard	Sleep request	SLEEPA
RATS		Yes	Standard	Activation of T4T	RATS
	ATS	Yes	Standard	T4T Response	ATS

Table 2.1: Commands for NFC-A

b8	b7	b6	b5	b4	b3	b2	b1	Meaning
x								Undefined
	x	x						00b Type 2 Tag 01b Type 4 Tag 10b NFC-DEP 11b Type 4 Tag and NFC-DEP
			x	x				Undefined
					x			1b NFCID1 not complete 0b NFCID1 complete
						x	x	Undefined

Table 2.2: The NFC-A SEL_RES (selection response) used at the end of the anti-collision activity contains information on the listener's capabilities.

NFC-A specifies a minimum and maximum timing between the end of a poll frame and the end of a listen frame. This time is called Frame Delay Time (FDT). For poll frames containing ALL_REQ, SENS_REQ, SDD_REQ, or SEL_REQ, the minimum FDT is also the maximum FDT. This means that frames from the listener must be sent at a specific point in time $FDT_{max} = FDT_{min}$ with a tolerance of $-1/f_c$ to $400\,\mathrm{ns} + 1/f_c$, i.e. $-74\,\mathrm{ns}$ to $474\,\mathrm{ns}$, where f_c is the carrier frequency of $13.56\,\mathrm{MHz}$. The value of the FDT depends on the bit duration $bd \approx 9\,\mu\mathrm{s}$ for $106\,\mathrm{kbit/s}$ and can be calculated with the following formula:

$$FDT_{min} = FDT_{max} = 9bd + 20/f_c \approx 86\,\mu\mathrm{s}$$

Taking the tolerance into consideration, the maximum value for the FDT is therefore $\approx 87\,\mu\mathrm{s}$ [20]. For all other frames, i.e., those not part of the anti-collision activity, the listener has no maximum timing constraints.

The maximum time between two poll frames is not specified. The time from the end of a listen frame to the start of a poll frame has no specified maximum time, either.

Activity

Anti-Collision The poller first sends an ALL_REQ command. All devices in the field respond with a SENS_RES. The SENS_RES contains the length of the NFCID1. If there is at least one SENS_RES, an SDD_REQ command is sent querying the unique NFCID1 of all devices. The NFCID1 is an ID with a length of 4, 7, or 10 bytes identifying an NFC-A listener device. For an NFCID1 longer than 4 bytes, the ID is split up into multiple cascade levels. Each cascade level has an optional 1-byte cascade tag and 3 or 4 bytes of the NFCID1.

All listener devices respond now with a SDD_RES that contains the bytes of the current cascade level of their NFCID1. If there is a collision in the response, i.e., two or more devices respond to the poller with different bytes of their NFCID1 for the current cascade level, the poller is able to detect this collision by receiving two opposing bits. If a collision occurs, the valid number of already received bits is noted, and the remaining bits are requested again. This is done by specifying the already valid bits in a SDD_REQ. Only listener devices with these bits will now respond to this SDD_REQ.

If there is another collision, the same procedure is repeated. With this procedure, the poller is able to systematically identify the NFCID1 of one listener device. At least one bit is always guaranteed to be resolved with each iteration.

Without a collision, a SEL_REQ is sent, and the listener device responds with a SEL_RES that contains its higher-level capabilities. The bit layout of a SEL_RES can be seen in Table 2.2. If there are still parts of the NFCID1 missing (the NFCID1 has multiple cascade levels for 7 or 10 byte lengths), the SDD_REQ is sent again for a higher cascade level until the entire NFCID1 has been received by the poller without collisions. After the final SEL_REQ has been exchanged with the listener, the listener finds itself in the selected state.

More listener devices can be resolved by putting the current device to sleep by sending a SLP_REQ, and then sending a SENS_REQ to query more devices. Only devices that are not sleeping respond to the SENS_REQ. Otherwise, the anti-collision activity ends here.

Device Activation Depending on the higher-level technology detected earlier, the poller sends

1. a Request for Answer To Select (RATS) for ISO Data Exchange Protocol (ISO-DEP) (used for T4Ts),
2. a ATR_REQ for NFC-DEP Peer-to-Peer (P2P) communication or
3. nothing for Type 2 Tags (T2Ts).

2.4.2 NFC-B

NFC-B is most commonly found in e-passports and e-tickets. The NFC-B standard is derived from ISO/IEC 14443 Type B [13, 14, 18] and is described in the NFC Forum specifications [20, 24]. Compared to NFC-A, NFC-B is less common and not supported on all NFC tag readers and writers.

Command	Response	Purpose	ISO alias
ALLB_REQ		Probing for listeners	WUPB
SENSB_REQ		Probing for listeners	REQB
SLOT_MARKER		Select slot	Slot-MARKER
	SENSB_RES	Response from listener	ATQB
SLPB_REQ		Sleep request	HLTB
	SLPB_RES	Response to sleep request	HLTB
ATTRIB Command		Activation of T4T	ATTRIB Command
	ATTRIB Response	T4T response	ATTRIB Response

Table 2.3: Commands for NFC-B

Digital Protocol

Transmission Speed NFC-B only allows communication at $106\,\mathrm{kbit/s}$.

Frame Format NFC-B has only one frame format for communication. The EoD of a frame contains two CRC bytes calculated from the payload bytes.

Commands NFC-B has fewer commands than NFC-A. Table 2.3 shows the commands and their respective responses. The ISO/IEC 14443 standard uses a different terminology for some commands.

Timing The NFC-B protocol gives the Frame Waiting Time (FWT) as the maximum time between the end of a poll frame and the start of a listen frame.

There is no maximum time defined between two poll frames. Neither is there a maximum time between a listen frame and a poll frame.

Activity

Anti-Collision Fundamentally, NFC-B uses time slots for each listener device. Each device randomly selects a time slot at the start. The number of slots is initially set to 1.

At the start, ALLB_REQ is sent to all devices. If there is a SENSB_RES response in the current slot number, and it didn't cause a collision, save it as a device and potentially resolve more; otherwise, end the activity here. If there is a response with a collision or no response in the current slot, increment the slot number and query the corresponding device slot to the device with a SLOT_MARKER. If this is the last slot available, increase the number of slots, put the last device into sleep mode with a SLPB_REQ command, unless the maximum number of slots (16) has been reached, then the activity is resolved.

Command	Response	Purpose
SENSF_REQ		Polling for listeners
	SENSF_RES	Response to poller

Table 2.4: Commands for NFC-F

Activation If a Type 4 Tag (T4T) is about to be activated, an ATTRIB command is sent, specifying protocol information. The T4T responds with an ATTRIB response. T4Ts are the only higher-level application supported by NFC-B.

2.4.3 NFC-F

NFC-F is based on the Japanese FeliCa JIS X6319-4 standard [10]. It supports higher data rates than the other NFC technologies.

Digital Protocol

Transmission Speeds NFC-F supports data rates up to $848\,\mathrm{kbit/s}$. Commonly, speeds of $212\,\mathrm{kbit/s}$ or $424\,\mathrm{kbit/s}$ are found in real world applications.

Frame Format Each frame begins with a SoF and is then followed by bytes of data. The data starts with a Start of Data (SoD) that is a length byte, indicating the length of the payload + 1. After this header, the payload bytes follow. After the payload, an EoD with 2 bytes of CRC is added. The CRC is calculated over the SoD and the payload.

Commands NFC-F only has one command and one response. They can be seen in Table 2.4.

Timing NFC-F uses time slots for its anti-collision activity.

All time slots have the same length, and they are directly next to each other with no intermediate guard time. Frames that start in one time slot and end in the next time slot may be rejected by the poller. The maximum time between two poll frames and between a listen and a poll frame is undefined.

Activity

Anti-Collision Similar to NFC-B, NFC-F employs a time slot-based system for collision resolution; it queries every device in the proximity with a SENSF_REQ request. Every listener device then randomly selects a time slot to respond in. If the device limit set beforehand is still larger than the current amount of valid responses, another SENSF_REQ is sent, setting the number of time slots to the maximum of 16. The time slots are based on the initial sending of the request by the poller. This means timing is important to match the time slot deadlines.

Command	Response	Flag	Purpose
INVENTORY_REQ		INV_FLAG	Probing for listeners
	INVENTORY_RES	RES_FLAG	Response to poller

Table 2.5: Commands for NFC-V

The NFCID2 of the response can hint at NFC-DEP support. If this is the case, another SENSF_REQ is sent to the specific devices, but with NFC-DEP protocol information.

Activation If the protocol is NFC-DEP, then an ATR_REQ is sent, followed by an optional PSL_REQ. Otherwise, nothing is done.

2.4.4 NFC-V

NFC-V is based on the ISO/IEC 15693 standard for contactless vicinity objects [16][17][27]. With the correct hardware, this technology allows communication at greater distances, hence the term vicinity. Although transmission speeds are slower than those of the other NFC technologies.

Digital Protocol

Transmission Speeds The transmission speeds can range from $6\,\mathrm{kbit/s}$ to $26\,\mathrm{kbit/s}$. T5Ts use the higher data rate of $26\,\mathrm{kbit/s}$.

Frame Format NFC-V uses standard frames, isolated EoF frames, and special frames. A standard frame starts with a SoF, n times 8 data bits, and an EoF. It is used for data exchange. The data consists of the payload bytes followed by two CRC EoD bytes. Isolated EoF Frames are only made up of one EoF symbol. They are used to signify the start of a time slot. A special frame is a composition of a standard frame and an EoF, which are separated by a FDT.

Command Set NFC-V provides one request and one response command. The one-byte flags are sent with these commands and contain more detailed information for the protocol. The commands can be seen Table 2.5.

Timing For the time from a poll frame to a listen frame a minimum time of $\approx 319\,\mu\mathrm{s}$ has to be passed and a maximum FDT of $\approx 323\,\mu\mathrm{s}$ for Read-Alike commands and $\approx 20\,\mathrm{ms}$ for Write-Alike commands can not be exceeded.

In the anti-collision process, if no listen frame is received after an INVENTORY_REQ or isolated EoF frame was sent previously by the poller in a time of $\approx 474\,\mu\mathrm{s}$, a timeout event happens, and the next poll frame can be sent. In case of a reception of a listen frame, the poller waits at least $310\,\mu\mathrm{s}$ before sending the next poll frame. A maximum time is not defined.

Tag	Technology	Read Size	Write Size	ISO-DEP	Based On
T1T	A	8 bytes	8 bytes	no	
T2T	A	16 bytes	4 bytes	no	
T3T	F	varies	varies	no	
T4T	A, B	varies	varies	yes	
T5T	V	varies	varies	no	
M. Ultralight	A	16 bytes (more w. proprietary protocol)	4 bytes (more w. proprietary protocol)	no	T2T
M. Classic	A	16 bytes	16 bytes	no	
M. DESFire	A	varies	varies	yes	T4T

Table 2.6: Comparison of NFC tags. The read and write sizes relate to a single command exchange. The entire tag can be accessed by repeatedly sending read or write commands.

Activity

Anti-Collision The collision resolution uses time slots. Initially, one slot is used: an INVENTORY_REQ is sent, and listeners respond with their INVENTORY_RES. If no answer arrives or only one device responds, the activity concludes. Multiple devices cause a collision, and the collision information is stored.

A second INVENTORY_REQ with 16 time slots is sent. MASK_LEN and MASK_VAL are variables and set to 0 and empty, respectively. Listeners respond based on their UID, with the last 4 bits after MASK_LEN indicating the slot number. Devices without collisions are added to the known devices list; collisions are saved for later resolution. The slot number is incremented by sending an isolated EoF frame.

When slot 16 is reached, and devices remain unresolved, another INVENTORY_REQ is sent with a longer MASK_VAL (and larger MASK_LEN). Different UIDs cause devices to respond in different time slots, enabling systematic resolution.

Activation For the activation with NFC-V, no action is required. The activation is already solved with the anti-collision activity.

2.5 NFC Tags

An NFC tag is a device that can store information, typically formatted as an NFC Data Exchange Format (NDEF) message. The tag itself does not need a power source and can be accessed by a compatible NFC reader/writer. NFC tags are based on the passive communication mode, with a reader/writer acting as an active poller and the tag acting as a passive listener. Not all NFC chips can emulate tag functionality, but some hardware mentioned later can act as a tag. An overview of the different tags and their capabilities can be seen in Table 2.6.

There are five tag types that are standardized by the NFC Forum. Most commonly, type 2 and type 4 tags — both based on NFC-A — are seen in the consumer market.

The memory layout of each tag type is different, and each tag type has its own command set. Each type is built on top of distinct underlying NFC technologies. Hence, tag types have no compatibility, e.g., the procedure to interact with a Type 2 Tag (T2T) can not be used for a Type 3 Tag (T3T).

Tags have varying degrees of access control via read/write protection or encryption. Commonly, manufacturers such as NXP provide their own tags that are extensions with partial NFC Forum compatibility. Proprietary NFC tags provide their own command set and offer security features not outlined in the NFC Forum tag specifications.

This section will cover tag types 1 to 5 as well as the proprietary MIFARE Ultralight, MIFARE Classic, and MIFARE DESfire tags.

2.5.1 Type 1 Tags

Type 1 Tags (T1Ts) are the oldest tags specified by the NFC Forum [6]. They are based on NFC-A. They are no longer included in the current versions of the NFC Forum digital and activity technical specifications and can therefore be considered legacy. They are based on NFC-A. T1Ts have largely been replaced by type 2 and type 4 tags.

Unlike newer tags, T1Ts do not use an anti-collision procedure to activate them. This means that only one tag should be in the generated RF field. They are instead activated via a standard NFC-A SENS_REQ/SENS_RES exchange that is followed by a special RID command that is answered by the type 1 tag with an RID response. The RID response contains the tag's UID. If a collision occurs during this process, due to two or more tags responding with their respective UIDs, no tag can be activated.

The tag's memory layout is structured in a continuous sequence of blocks, each containing 8 bytes. A tag can either contain a fixed amount of 15 blocks or a dynamic amount of up to 256 blocks of memory. The application data is structured in Type-Length-Value (TLV) blocks.

2.5.2 Type 2 Tags

Type 2 Tags (T2Ts) are NFC tags based on NFC-A [7]. After the NFC-A anti-collision activity, the reader/writer can directly send T2T commands to interact with the tag.

The T2T memory is structured into blocks or pages of 4 bytes. The first 4 blocks are called reserved bytes as they are not intended to store actual data. The first 2 blocks and half of block 2 contain internal information totaling 10 bytes, that is, the UID of the tag and checksum information. For tags with 64 bytes of data, the memory layout is called static, and memory layouts with more than 64 bytes are called dynamic according to the specification.

The 2 lock bytes in block 2 are called static lock bytes and contain information about write access to only the first 64 bytes of the tag. If both are set to `0xFF`, the tag can no longer be written to. Once the bits are set to 1, they can no longer be set back to 0, so write access is permanently denied, and the tag is effectively read-only. For tags longer than 64 bytes, additional lock bytes

are placed after the end of the data area to define write access for other sections of memory inside the tag.

Block 3 is occupied by the Capability Container (CC). The first byte of the CC indicates the presence of an NDEF message. If byte is `0xE1`, the tag contains an NDEF message. Other values are used to indicate proprietary information. The second byte contains the version number of the tag. The next byte contains the memory size. This byte indicates the maximum size of the tag memory and not the size of the current data stored in the tag. The actual memory of the tag in bytes is the result of this number multiplied by 8. We therefore get a maximum memory size of 2048 bytes. The last byte of the CC indicates possible write access to the tag. In contrast to the two lock bytes discussed earlier, these bytes are only used for indication and don't necessarily reflect the real read or write locks of the tag.

After the 16 reserved bytes, the data area follows. The data area contains Type-Length-Value (TLV) blocks. The most relevant TLV block is the NFC Data Exchange Format (NDEF) Message TLV with a type of `0x03` containing the NDEF message. The NDEF message TLV is present in every type 2 tag. Other TLVs can indicate additional lock bytes.

After all other TLV blocks, the terminator TLV with `0xFE` is the last TLV and marks the end of the tag's payload. The rest of the data area is padded with null bytes. The memory layout can be seen in Figure 2.2.

To interface with the tag, three commands are provided by the type 2 tag specification: the `READ` command to read 4 blocks of data, `WRITE` command, to write one block of data and `SECTOR SELECT` to access higher memory areas, because the one byte identifying the block can only address memory up to 1024 bytes. A tag can have up to two sectors. Additionally, the tag responds with an `ACK` on success or a `NACK` on failure for `WRITE` commands.

At the end of all communication, the reader/writer can issue a `SLP_REQ` command that puts the tag into sleep mode. It needs to be woken up again according to the anti-collision activity outlined for NFC-A.

2.5.3 Type 3 Tags

Type 3 Tags (T3Ts) are exclusively used on top of NFC-F [8, 10]. Real-life variants are the proprietary Sony FeliCa tags. They are used in Asia for payment systems [37].

All type 3 tags contain read-only information about the manufacturer and the system. A T3T has multiple services that can be thought of as files. Each service has a 2-byte identifier and a 2-byte block count. A block has a size of 16 bytes.

With a Check command, multiple blocks can be read from different services by a reader. The same holds true for the Update command that enables block writing. A T3T can store an NDEF message inside an NDEF service.

NFC Type 2 Tag Memory Layout				
	Byte 0	Byte 1	Byte 2	Byte 3
Block 0	Internal 0	Internal 1	Internal 2	Internal 3
Block 1	Internal 4	Internal 5	Internal 6	Internal 7
Block 2	Internal 8	Internal 9	Lock Byte 1	Lock Byte 2
Block 3	Magic Number (0xE1)	Version Number	Memory Size	Read/Write Access
Block 4	Data	Data	Data	Data
...	...	...	...	...
Block k	Data	Data	Data	Data

Figure 2.2: The NFC Type 2 Tag memory is split up into blocks of 4 bytes. The blue section marks the internal bytes, the yellow section is the lock bytes, and the green section highlights the Capability Container (CC). The data area starts at block 4 and ends at block k.

Command APDU Header				Command APDU Body (optional)		
CLA	INS	P1	P2	Lc	Command Data	Le
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Lc Bytes	Last Byte

Figure 2.3: The command APDU has an optional body and is always sent by the poller.

Response APDU Body	Response APDU Status	
Response Data	SW1	SW2
Lc Bytes	2 Bytes	

Figure 2.4: The response APDU is the answer of the tag and contains data and two mandatory status bytes. It is always sent by the tag in response to a command APDU.

SoD			Payload	EoD
PCB	[DID]	[NAD]	[Information]	CRC
Byte 1	Byte 2	Byte 3	Payload Bytes	Last 2 Bytes

Figure 2.5: The ISO-DEP block format contains an ID inside the optional DID field and a node address inside the optional NAD field that is not supported by most cards. The payload information is also optional.

2.5.4 Type 4 Tags

Type 4 Tags (T4Ts) are based on NFC-A and NFC-B and implement the entire stack of ISO/IEC 14443 specifications for identification cards, including the unique ISO-DEP transport layer [9, 15]. For the application layer, they use Application Protocol Data Units (APDUs). They are more elaborate than T2Ts, which only conforms to parts 1 to 3 and doesn't adhere to the transport layer defined in part 4 of the ISO/IEC 14443 standard. MIFARE DESFire tags comply with the T4T protocol.

After the normal NFC-A or NFC-B anti-collision activity, the device activation for a T4T involves the reader/writer sending a Request for Answer To Select (RATS) for NFC-A or an ATTRIB command for NFC-B. These commands include the maximum frame size. The T4T responds with an Answer To Select (ATS) for NFC-A or an ATTRIB response for NFC-B. When device activation is finished, further data exchange works over ISO-DEP.

The tag memory is a virtual file system. Every T4T contains an NDEF application with a Capability Container (CC) file and an NDEF file. The CC file contains information on the maximum payload size for commands, the maximum NDEF file size, and information on read/write permission. The NDEF file starts with a 2-byte nlen header that holds the current length of the NDEF message inside the file. After the nlen header, the actual NDEF message follows. Unlike other tag types, the NDEF message inside the tag can't be interrupted by metadata such as keys.

The actual commands are encapsulated in Application Protocol Data Units (APDUs). They are defined in the ISO/IEC 7814 standard for identification cards and integrated circuit cards [19]. The reader and writer of a tag sends command APDUs and the tag responds with response APDUs. The two APDU types can be seen in Figure 2.3 and Figure 2.4. The CLA field is the class and is set to 0 for non-proprietary commands. The INS field contains the instruction. The Lc field indicates the length of the payload, and the Le field is the expected length of the response. The two SW bytes indicate the status of the operation.

The standard contains a variety of instructions for APDUs where only three are used for T4Ts.

1. READ_BINARY to read files requires the Le field,
2. UPDATE_BINARY to write data to files; it requires the Lc field and a payload
3. SELECT to select which file to operate on.

For all commands, a successful operation is indicated with a status value of `0x9000` inside the response.

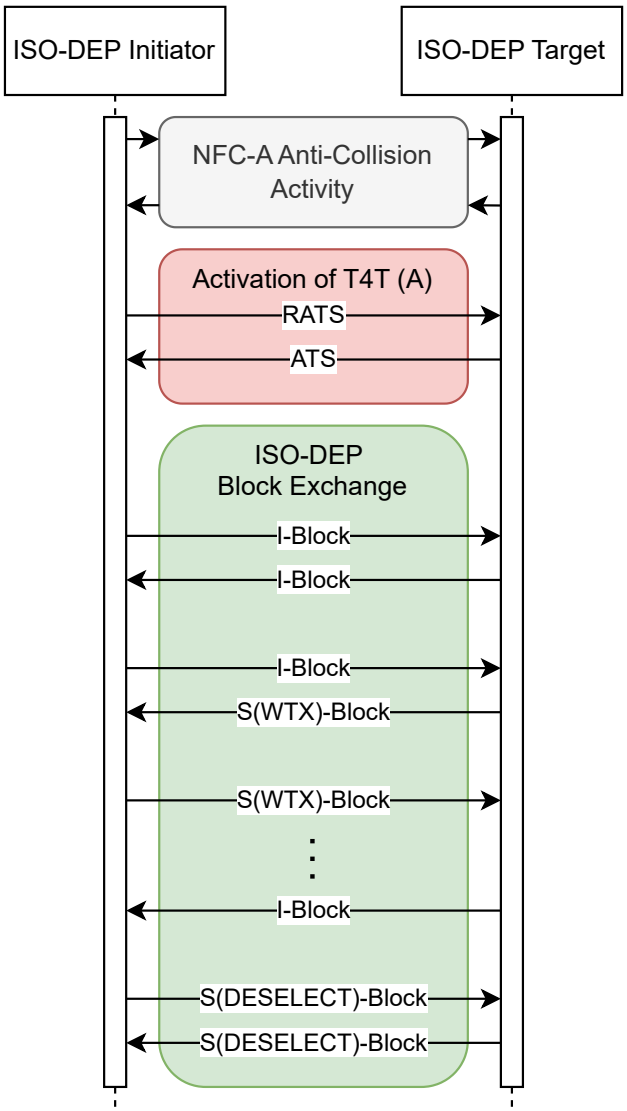

Figure 2.6: The initiator or poller starts the anti-collision activity for NFC-A with the target or listener. In this case, the target acts as a T4T. Afterwards, the RATS/ATS exchange happens to activate the tag. The block starts with an I-Block exchange. After the first exchange, the target requires more time to process the next I-Block, and the S(WTX) is sent to extend the time. The initiator acknowledges this extension by responding with its S(WTX). The I-Block is then sent at a later time. At last, the initiator deselects the tag with an S(DESELECT)-Block

ISO-DEP

ISO Data Exchange Protocol (ISO-DEP) is the transport layer only used for T4Ts. This means it only works with NFC-A and NFC-B. It was originally defined in part 4 of the ISO/IEC 14443 standard [15] and later integrated into the NFC Forum Digital Protocol Specification [20]. ISO-DEP is a half-duplex block transmission protocol that works the same way for NFC-A and NFC-B. Half-duplex means that only one device can send at any given point in time. An example exchange can be seen in Figure 2.6.

ISO-DEP defines its own block format. Every ISO-DEP block starts with a SoD. This SoD must always contain a Protocol Control Block (PCB) and optional bytes for the device ID. The PCB encodes the block type, message chaining, the existence of optional bytes, and the 1-bit block number. The SoD is followed by the payload.

Every block ends with a 2-byte CRC. The CRC calculation depends on the NFC technology used. Three different block types are defined for ISO-DEP:

1. The I-Block (information block) is used for exchanging data between poller and listener. I-Blocks can be chained to circumvent the maximum frame size established earlier.
2. The R-Block (response block) is used to acknowledge (ACK) received I-Blocks indicating chaining from the poller. A not acknowledge (NAK) can also be encoded in case of protocol errors or a timeout. The R-Block does not contain a payload.
3. The S-Block (supervisory block) can either be used for Wait Time Extension (WTX) if the listener needs more time to process the information, or it can be used from the poller to deselect the tag.

Protocol Operation The reader/writer always sends the first I-Block. If the I-Block indicates chaining, then the tag responds with an R(ACK); otherwise, it responds with its own I-Block.

Both devices keep track of a 1-bit block number. The reader/writer has it initially set to 0 and the tag to 1. The block number is toggled before sending the next block when a valid I-Block or an R(ACK) is received. This guarantees that the block number of received blocks alternates between 0 and 1 on both ends of a valid ISO-DEP communication. If a block number violates this pattern (two consecutive 1 or 0-bit block numbers), a retransmission is done.

2.5.5 Type 5 Tags

Type 5 Tags (T5Ts) are based on NFC-V. They are used in logistics and inventory management.

The tag memory is organized in blocks of 4, 8, 16, or 32 bytes. The first block of the tag memory contains the CC. The CC stores information about access control and the length of the tag memory. The maximum tag memory can't exceed 65536 bytes.

The T5T specification provides a READ_SINGLE_BLOCK command to read one block of the tag's memory. READ_MULTIPLE can read an arbitrary number of blocks at once. WRITE_SINGLE_BLOCK and WRITE_MULTIPLE function analogously. The tag can be put to sleep with a SLPV_REQ. This can be used to interact with a different T5T in the vicinity.

2.5.6 MIFARE Ultralight

MIFARE Ultralight (MFU) tags are based on T2Ts. They are manufactured by NXP. They can be activated according to the NFC-A specification, and interfacing with the tags follows the type 2 tag specification.

There are multiple MIFARE Ultralight tags available, like Ultralight C, Ultralight AES, and Ultralight EV1. They differ in their level of protection, with stronger encryption offered by some tag variants. The way to interact with them doesn't change depending on the variant of the Ultralight tag, so the individual variants aren't described further.

After the data area of the tag, special configuration pages exist on the tag to indicate a variety of configuration options. The AUTH0 byte indicates the page that is the start of the encrypted memory area. The AUTHLIM byte indicates the maximum number of negative password verification attempts. The configuration area also stores the password used for authentication. Password access can be enabled by setting the PROT field to 1. By default, password access is disabled.

On top of the NFC Forum-defined commands, MIFARE Ultralight tags use a set of extended commands that can be used for advanced access. With the FAST_READ command, the reader/writer can specify a range of pages that will be read from the tag at once. For encryption, the PWD_AUTH command takes a 4-byte password for the tag. Nonetheless, the MIFARE Ultralight tag can be treated like a T2T as long as password access is disabled.

2.5.7 MIFARE Classic

MIFARE Classic (MFC) tags use NFC-A, and they are manufactured by NXP.

The memory layout is different from a T2T as memory is structured into sectors. A sector can either be small, containing 4 blocks, or large, with 16 blocks. Each block contains 16 bytes of data.

The first block of sector 0 always contains the MIFARE Application Directory (MAD). This block contains either 4 or 7 bytes of the tag's UID. The other bytes are manufacturer data. Depending on the access bytes, blocks 1 and 2 of sector 0 can also be used for the MAD. If not, they can be filled with regular data. The MAD stores Application Identifiers (AIDs) for sectors. They describe the data stored in those sectors. The list of AIDs is curated by NXP.

Regular data blocks can contain TLV data just like T2Ts. The last block of each sector is a trailer block that contains a key A and a key B used for authentication. The access bytes define whether key A or key B needs to be used to access the data blocks in this sector. Access to specifically the keys or the access bytes can be controlled as well. Access to blocks can be forbidden entirely, effectively locking the tag. The trailer blocks contain no user data.

There are two MIFARE Classic sizes available: MIFARE Classic 1K tags with 1024 bytes split up into 16 small sectors and MIFARE Classic 4K tags with 32 small and 8 large sectors, leading to a total size of 4096 bytes. The memory layout of a MIFARE Classic 4K tag can be seen in Figure 2.7.

Figure 2.7: This is the memory layout of a MIFARE Classic 4K tag. The top numbers indicate the byte, and the left side shows the sector and the block. The yellow section contains the UID and the manufacturer data. The trailer blocks contain the key A in blue, 4 access bytes in red, and a key B in green.

Every time a new sector is accessed, an AUTH command needs to be sent to the tag. There are two AUTH commands for keys A and B. The AUTH command consists of a block number from the sector and a checksum according to NFC-A. Both sides now perform a handshake where a shared secret between the two devices is established using the correct key and the UID of the tag. The actual encryption is based on the proprietary CRYPTO1 cipher. This handshake process is subject to a timeout of 2 milliseconds. The exchange of data during the handshake process uses NFC-A short frames.

Due to the proprietary nature of the CRYPTO1 cipher, the timeout requirements, and the unusual framing, most NFC devices implement this handshake in hardware.

Accessing the tag data can be done with a READ command. The READ command returns an entire block. The WRITE command writes one block, and it involves a two-part procedure where first the block address is sent by the poller, and only after an ACK by the tag, the actual bytes can be sent. Both commands require a secure channel established earlier with the AUTH command.

2.5.8 MIFARE DESFire

MIFARE DESFire tags by NXP are based on T4T and ISO-DEP and therefore use APDUs for communication. MIFARE DESFire tags come in different sizes and in different security modes: EV1, EV2, and EV3. They support encryption, but for the purposes of this work, they can be treated like an NFC Forum T4T as long as no authentication is being used.

2.6 Peer-to-Peer Communication

Unlike previous sections, Peer-to-Peer (P2P) communication allows for bidirectional exchange of data. The communication takes place between only two devices, where both are considered to be peers. There are two variants of P2P communication: the SNEP as an application protocol built on top of the LLCP and NFC-DEP. The other is the TNEP using the tag specifications for P2P communication. All P2P communication still involves an NFC poller and a listener.

2.6.1 NFC Data Exchange Protocol

The NFC Data Exchange Protocol (NFC-DEP) is the lower-level protocol for bidirectional data exchange. It was originally defined in the ISO/IEC 18092 standard [23] and then later taken into the NFC Digital Protocol Specification [20] and the NFC Activity Specification [24].

The NFC-DEP is activated after the anti-collision loop of either NFC-A or NFC-F. The poller activates the listener with an Attribute Request (ATR_REQ). The ATR_REQ contains a unique 10-byte NFCID3 of the poller and the bit rate supported in sending and receiving directions by the poller. The maximum payload is also specified as either 64, 128, 192, or 254 bytes. Additionally, a few optional and historical bytes are specified, which are not required to understand the protocol. The listener responds with an Attribute Response (ATR_RES) that, in turn, is made up of

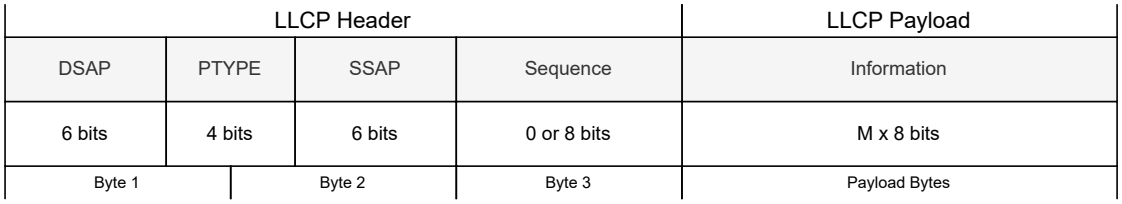

Figure 2.8: The LLCP PDU Format with Destination Service Access Point (DSAP), PDU Type (PTYPE), Source Service Access Point (SSAP), optional sequence number and payload information.

its NFCID3 and bit rate capabilities. The listener can set the Waiting Time (WT). This determines the maximum time the poller waits for a response before a timeout error is issued.

During further exchanges, the initiator can change previously defined communication parameters by sending a Parameter Selection Request (PSL_REQ). The listener will respond with a Parameter Selection Response (PSL_RES).

For actual data exchange, the DEP_REQ is sent to the listener. The listener responds with a DEP_RES.

To put the listener into a sleep state, the poller sends a Deselection Request (DSL_REQ). The listener must respond with a Deselection Response (DSL_RES). Afterwards, the listener can be activated again.

To put the listener into an idle state, the poller sends a Release Request (RLS_REQ). The listener must respond with a Release Response (RSL_RES). In the idle state, the anti-collision loop must be performed again to activate the listener according to NFC-A or NFC-F.

NFC-DEP uses blocks encapsulated in DEP_REQ/DEP_RES exchanges. The structure of these blocks and their function are similar to those described in ISO-DEP used for type 4 tags. A few important changes must be noted, however:

- the block number used for error detection consists of 2 bits instead of 1; this number is incremented after a successful reception,
- the Wait Time Extension (WTX) used for ISO-DEP in S-Blocks is called Response Timeout Extension (RTOX),
- deselection is not done with the S-Block; instead, the S-Block can be used for Attention (ATN) to perform a presence check. All addressed devices must respond with ATN in return.

2.6.2 Logical Link Control Protocol

The Logical Link Control Protocol (LLCP) is built on top of NFC-DEP to guarantee reliable data exchange. NFC-DEP can be understood as the MAC layer for LLCP. The LLCP requires the underlying layer to be the NFC-DEP. This means that the protocol still relies on a request/response mode of communication, which is then abstracted away to function like a proper peer-to-peer protocol.

The data is encapsulated in Protocol Data Units (PDUs). A PDU has a header and an optional payload. The header contains the DSAP, the PDU type, and the SSAP. Depending on the PDU type, the PDU header can include a sequence number field. The structure of a PDU can be

seen in Figure 2.8. The SSAP and DSAP can be understood as unique identifiers of services running on the device. The LLCP can be split up into the lower sub-layer of link management and the higher sub-layer of the transport mode.

Link Management

The link management ensures protocol-compliant link-level communication with the other device. This means that all PDU types used for link management don't contain a DSAP or a SSAP as they are not service-specific.

During the data exchange, any device can send a Parameter Exchange (PAX) PDU to change the link's parameters, like the Link Time Out (LTO), the version number, or the Maximum Information Unit (MIU) - among others. The parameters can also be transmitted as optional bytes as part of the ATR_REQ/ATR_RES pair outlined in the NFC-DEP.

The version number is split up into a minor and a major version. Both devices need to have the same major version number. The version number must be exchanged at the start of the link activation as a parameter.

Each device must send data before the LTO has passed. If no data is present, a Symmetrical (SYMM) PDU is exchanged that contains no data but keeps the communication symmetrical. Multiple PDUs can be sent in an Aggregated (AGF) PDU.

Transport Mode

The transport mode can be either connectionless or connection-oriented. An LLCP-capable NFC device must be able to handle both transport modes.

Connectionless Transport The connectionless transport mode for LLCP is simpler than the connection-oriented mode, but doesn't guarantee that data will be passed to the higher-level application.

No connection handshake is necessary, and after the link has been established, Unnumbered Information (UI) PDUs can be exchanged. A UI PDU has no sequence number but a DSAP and SSAP field.

Connection-oriented Transport The connection-oriented transport mode for LLCP requires connection establishment with individual acknowledgments for each frame passed to the higher-level application. This ensures that no data is dropped. A detailed explanation of all PDU types is not required as the connection-oriented transport mode won't be implemented.

2.6.3 Simple NDEF Exchange Protocol

The Simple NDEF Exchange Protocol (SNEP) is built on top of the LLCP and enables the exchange of NDEF messages between two devices. The SNEP is request-response-based, stateless, and defines a client and a server. The SNEP protocol defines methods to query NDEF messages with GET or push them with PUT. SNEP allows for message fragmentation to overcome constraints by the underlying LLCP transport layer.

2.6.4 Tag NDEF Exchange Protocol

The Tag NDEF Exchange Protocol (TNEP) is defined by the NFC Forum. Unlike previous NFC P2P protocols, it is built on top of the NFC Forum tag specification and therefore doesn't require the stack outlined for the SNEP. The NFC devices must service special NDEF records that contain information for data exchange.

2.7 NFC Data Exchange Format

The NFC Data Exchange Format (NDEF) is a data format specifically designed for NFC by the NFC Forum [1]. It is used to encapsulate data for NFC tags and P2P communication alike.

Each NDEF message is made up of at least one record. A record has a header that encodes whether the record is the start or the end of an NDEF message, the type, the optional ID, and the payload length. A record can be either short or normal as defined in the first byte of the header. A short record has a length field of one byte — a normal record has a length field of 4 bytes.

The possible types for an NDEF message are defined in NFC Forum Record Type Definitions (RTDs). Common record types are text, URI, and MIME. After the header, a payload is included that conforms to the record type.

2.8 NFC Controller Interface

The NFC Controller Interface (NCI) is specified by the NFC Forum [25]. It provides a unified way to interact with a compatible NFC Controller (NFCC) from a Device Host (DH). Both together make up an NFC Forum Device. The NCI can be used to interact with NFC Execution Environments (NFCEEs) in which applications are executed or with a Remote NFC Endpoint via an RF protocol. An NFC Forum Device using the NCI but without an NFCEE can be seen in Figure 2.9. The NCI specifies a routing table on the NFCC that determines where data is forwarded to in case of multiple NFCEEs.

The NCI supports logical connections that can be used to address data to different endpoints, i.e., an NFCEE or a Remote NFC Endpoint. This feature is optional, as the static RF connection used for RF communication is always initialized with Connection Identifier (Conn ID) 0. Flow control is supported in the direction from the DH to the NFCC.

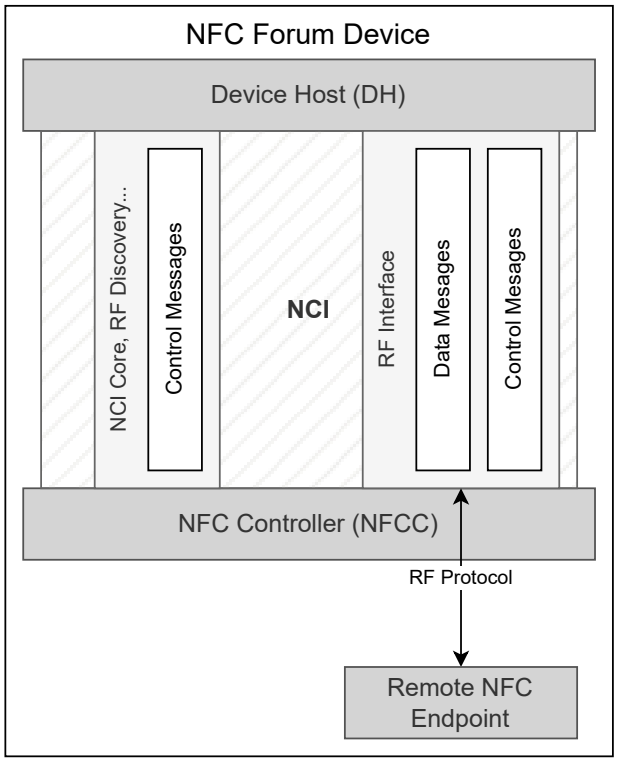

Figure 2.9: The DH and the NFCC communicate through different interfaces requiring different protocols. The NCI might have additional interfaces not displayed here.

Control Packet Header						Control Packet Payload
MT	PBF	GID	RFU	OID	Payload Length (L)	Payload
3 bits	1 bit	4 bits	2 bits	6 bits	8 bits	L x 8 bits
Byte 1			Byte 2		Byte 3	Payload Bytes

Figure 2.10: The NCI Control Packet has a Message Type (MT), a Packet Boundary Flag (PBF), a Group ID (GID), an Opcode ID (OID), a payload length byte, and the payload.

Data Packet Header						Data Packet Payload
MT	PBF	Conn ID	RFU	CR	Payload Length (L)	Payload
3 bits	1 bit	4 bits	6 bits	2 bits	8 bits	L x 8 bits
Byte 1			Byte 2		Byte 3	Payload Bytes

Figure 2.11: The NCI Data Packet has a Message Type (MT), a Packet Boundary Flag (PBF), a Connection ID (Conn ID), Credits (CR), a payload length byte, and the payload.

Hardware	NFC-A		NFC-B		NFC-F		NFC-V	
	Poll	Listen	Poll	Listen	Poll	Listen	Poll	Listen
nRF52840		x						
PN532	x	x	x		x	x		
PN7160	x	x	x	x	x		x	
ST25R3916B	x	x	x		x	x	x	

Table 2.7: The overview is split into poll and listen mode for each NFC technology. The hardware can be put into these modes with the correct configuration. The user might need to implement parts of each activity in software.

Hardware	NCI	IRQ	MFC	Serial Interface			In Hardware	
				SPI	I2C	UART	ISO-DEP	NFC-DEP
nRF52840		x						
PN532		x	x	x	x	x	x	x
PN7160	x	x	x	x	x		x	x
ST25R3916B		x		x	x			

Table 2.8: Overview of features for each NFC hardware chipset. The supported serial interface also depends on the board that is used. Not all boards support all serial communication types of the chipset. MFC stands for MIFARE Classic and symbolizes the support for the proprietary protocol.

The NCI defines control and data packets. The former are used to change the state and configuration of the NFCC, e.g., to put the NFCC into discovery mode, while the latter are used for data exchange. All packets start with a 3-byte packet header. The packet header always contains the Message Type (MT) and a Packet Boundary Flag (PBF). The MT distinguishes between data packets and control packets. If the PBF is set to 1, the packet is part of a segmented message, and this packet is not the last segment.

Control packets have a Group Identifier (GID) and an Opcode Identifier (OID) that identifies the purpose of the control message. The control packet layout can be seen Figure 2.10.

Data packets must contain the Conn ID indicating the logical connection to which the data belongs. The Credits (CR) are used for optional flow control. The data packet layout can be seen in Figure 2.11.

The NCI Core is the most essential part of the NCI as it is used to initialize and configure the device. The RF Discovery is used to make the NFCC poll for possible listener devices. The RF Interface conveys information that is then sent with the previously configured RF protocol to the Remote NFC Endpoint.

2.9 Hardware

The hardware needs to be capable of NFC and must provide some way to interact with it — either with a serial transport protocol or by in-memory editing. Some NFC hardware, like the nRF52840, features a built-in NFC peripheral that can be accessed by writing to and reading from registers.

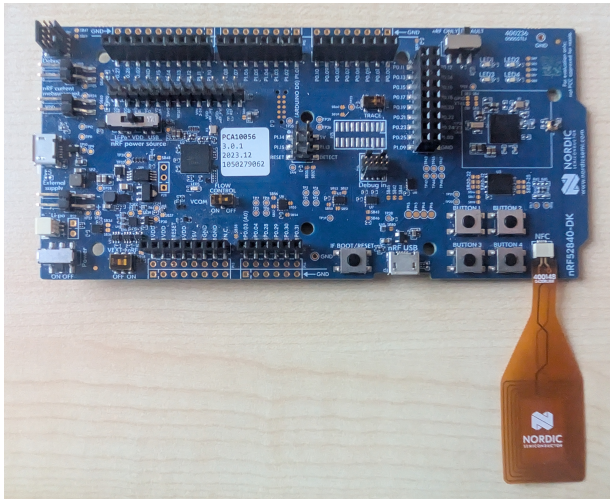

Figure 2.12: An image of the nRF52840 development kit used for testing. The orange NFC antenna needs to be attached to ensure NFC functionality.

The PN532 and the PN7160 sport their own firmware-specific commands. These commands must be used to configure the device into poll and listen modes. An overview of which NFC modes are supported by the hardware can be seen in Table 2.7.

2.9.1 nRF52840

The Nordic Semiconductor nRF52840 is a SoC with an ARM Cortex-M4 Microcontroller Unit (MCU) and is supported by RIOT OS [33]. The MCU has a built-in NFC peripheral that requires attaching an NFC antenna to GPIO pins 0.09 and 0.10.

The nRF52840 only supports NFC-A in listener mode for tag emulation. Unlike other NFC hardware, the communication with the NFC peripheral is done in memory. An IRQ can be issued on incoming NFC events that need to be handled by the driver. The nRF52840 has no hardware support for protocols like NFC-DEP and ISO-DEP. These must be implemented in software. The nRF52840 development kit is used for testing, and it can be seen in Figure 2.12.

2.9.2 PN532

The PN532 by NXP is an NFC-IC capable of reader/writer mode, tag emulation, and P2P communication [2]. For that, it offers a set of high-level commands that can be sent over I2C, UART, or SPI from the host device.

It provides support for NFC-A tag emulation as well as reader/writer mode. For NFC-B, it can only take over the reader/writer part of the communication. It is compatible with the NFC-F reader/writer mode and tag emulation mode. Additionally, it covers proprietary functionality of the MIFARE Classic tags.

For low-level access, the PN532 also provides the ReadRegister and WriteRegister commands. With them, low-level access to the hardware is given to the user, with the added complexity of taking care of frame formatting, checksum calculation, and overall configuration in software on the host device. The PN532 can be enabled to use an IRQ pin that is pulled low when the

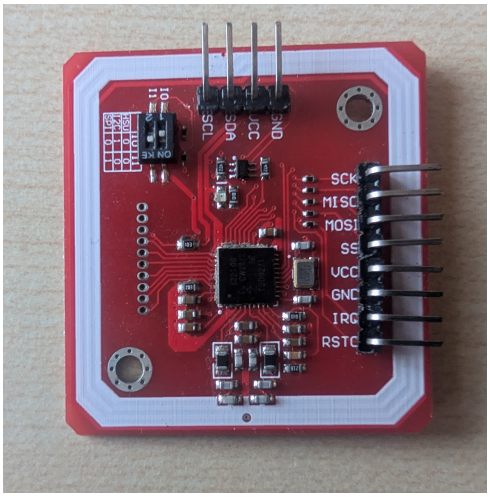

Figure 2.13: An image of the PN532 board used for testing. This board has the same name as the chip and is made by Elechouse.

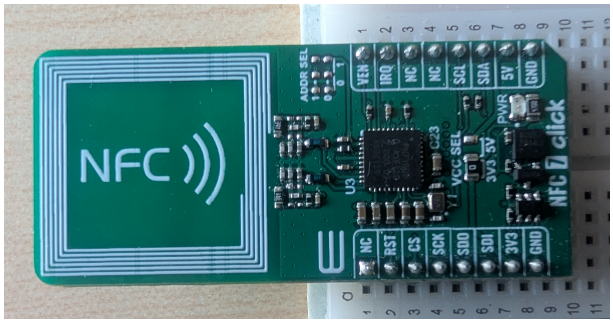

Figure 2.14: An image of the PN7160 board used for testing. This is the NFC 7 Click by Mikroe.

PN532 is ready to send data to the host. With no IRQ enabled, the host has to constantly poll the serial connection for data. When polling a MIFARE Classic tag, the PN532 takes care of the proprietary authentication process when provided with a key.

Not all PN532 variants are identical; some do not support all modes advertised by NXP. In particular, the listener functionality of the PN532 does not work on some variants of the PN532 chipset. The PN532 board provided by Elechouse does not work in listen mode[1]; it can be seen in Figure 2.13. The board can be changed into the different serial modes by adjusting two small switches.

2.9.3 PN7160

The PN7160 chip by NXP provides support for most NFC technologies [31]. It is compliant with the NCI and provides an SPI and I2C interface. A GPIO pin can be configured to trigger an IRQ when there is data available on the NFCC.

The PN7160 restricts its interface usage in tag emulation mode: the tag emulation mode can only be used with the ISO-DEP interface. This means that the PN7160 always emulates an ISO-DEP target and can't act as an NFC T2T.

[1] https://www.elechouse.com/product/pn532-nfc-rfid-module-v4/

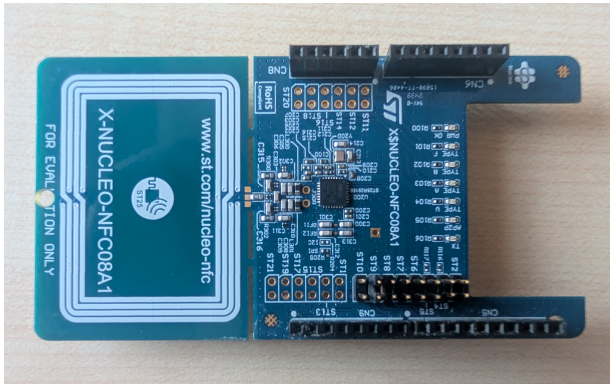

Figure 2.15: An image of the ST25R3916B board used for testing. This is the X-NUCLEO-NFC08A1 by ST Microelectronics.

For polling, the device provides a special MIFARE Classic tag interface (TAG-CMD interface) that takes care of the proprietary authentication handshake. To interact with NFC tags, the NDEF interface can be used to directly access the NDEF message of a tag. This provides a tag-agnostic way to interface with the NDEF contents of a tag without implementing the tag protocol on the host. The board used for testing is the NFC 7 Click[2] by Mikroe in the SPI configuration, and it can be seen in **??**.

2.9.4 ST25R3916B

The ST25R3916B by ST Microelectronics is an NFC RF frontend [32]. In contrast to the other NFC hardware, it does not provide a set of high-level commands. The communication with the host is done with SPI or I2C. The device needs to be configured entirely by writing into specific registers. The software is fully responsible for handling all NFC activities and protocols.

The RF frontend features an internal FIFO queue that must be filled with data before transmission. For the reception of data, the FIFO is filled with the data by the ST25R3916B. The host must issue commands to start transmission and allow reception of data. A register holds the number of bytes that are currently in the FIFO queue.

The RF frontend features an IRQ GPIO pin that is triggered for all sorts of events. The events that trigger an event must be defined by the user. These events can include a successful transmission or reception of NFC data, a timeout, and bit collisions. This covers the basic operations of the ST25R3916B. The board used for testing is the X-NUCLEO-NFC08A1[3]. An image can be found in Figure 2.9.4.

[2] https://www.mikroe.com/nfc-7-click-spi
[3] https://www.st.com/en/evaluation-tools/x-nucleo-nfc08a1.html

3 Related Work

This chapter provides an overview of readily available NFC libraries for the hardware mentioned in the previous chapter. The comparison part of this chapter is limited to the most feature-rich options that will be compared at the end. For the sake of better comparison of later design decisions, it will also include a breakdown of the current state of NFC in RIOT OS.

The four libraries mentioned here include two user-space libraries, *libnfc* and *nfcpy*, that can be acquired by using the package manager on Ubuntu and other Linux distributions. The *ST25 library* and the *nRF Connect SDK* are suited for IoT devices. The four libraries chosen are followed by a comparison of their protocol and NFC device support.

The Linux and Android NFC stacks have NFC libraries, and they support a wide range of devices and protocols, but are heavily integrated with the rest of the kernel. The device drivers included are not compatible with the breakout boards used for IoT applications. They target mobile devices instead. For this reason, the Linux NFC stack is not considered.

3.1 Libnfc

Libnfc is an NFC library written in C and supports NFC-A, B, and F [28]. It only contains driver code for the PN53X family. The PN71XX is also supported if an additional library called *libnfc-nci* is linked. Communication between the host and the NFC device can be done over SPI, I2C, USB, or UART.

The code is split into parts, one belonging to the PN53X chipset. The chipset contains functions to interact with the device according to the firmware specification of the PN53X, such as sending or receiving data or setting the device into a poll or listen state. This chipset is then used by individual drivers for different device types that make use of this chipset.

Libnfc defines an `nfc_driver` struct that holds the driver name and function pointers to interact with the driver. An additional `nfc_device` struct holds information about framing and protocol capabilities and a pointer to an `nfc_driver` struct. The library uses the C function `malloc` to allocate dynamic memory for the `nfc_context` that holds internal options and configuration about the library. This dynamic memory allocation is unsuited for the IoT.

The high-level code is split up into `utils` and `examples`. The directories contain fully-functional code that makes use of the underlying library. The code is not exposed in header files. Strictly speaking, the functionality to read and write tags is not part of the core library and must be implemented separately, as is done with the aforementioned utilities and examples provided as part of the libnfc repository.

The library supports reading and writing to T2T, T3T, T4T, MIFARE Ultralight tags, MIFARE Classic tags, and T1Ts. Example code is provided to make the PN53X act as a T2T, T3T, or T4T. NFC-DEP is supported in poller and listener mode, so P2P is possible. The LLCP is not supported.

3.2 Nfcpy

Nfcpy is an NFC library written in Python [22] and supports NFC-A, B, F as well as NFC-DEP with the LLCP and the SNEP for P2P communication. The supported tag types are 1, 2, 3, and 4. The library can solely be used with chipsets of the PN53X family and devices using this chipset. In particular, the PN531, PN532, and PN533 chipsets are supported.

The code follows object-oriented programming principles. An abstract `Device` class and `Chipset` class are defined that provide methods that need to be implemented by subclasses. Each `Device` subclass has its own implementation of the necessary technology-specific functions to act as a poller or listener and for data exchange. For the low-level serial communication, the `Chipset` class is used. This `Chipset` class contains firmware-specific code for the communication with the chipset. `Chipset` and `Device` parent classes for the PN53X family are defined that handle common functionality. Specific variants of the PN53X inherit from these classes, thereby increasing modularity.

The only supported transport protocol between the host device and the PN53X is UART, despite the chipset supporting I2C and SPI, too. The application layer features an implementation of the NFC-DEP, LLCP, and SNEP for P2P communication. This includes support for connectionless and connection-oriented modes of LLCP communication. The SNEP has multithread support and is split up into client and server code.

An abstract `Tag` class to interact with NFC tags as a poller is defined, and it contains an `NDEF` class. The `Tag` class handles remote reading and writing operations onto the tag. The `NDEF` class represents the NDEF data present on the tag. The user can directly access the NDEF message and overwrite it, if necessary. T1T and T2T possess a special `MemoryReader` class to view the tag as one continuous byte stream. T3T and T4T do not have this helper class. Proprietary variants of T2T by NXP, like MIFARE Ultralight tags, are supported by nfcpy. MIFARE Classic tags are not supported. A framework for tag emulation is only available for T3T, and the user still needs to register services, so the API does not expose simple tag emulation of an NDEF message.

Nfcpy does not include an NDEF library for encoding, decoding, and parsing. NDEF messages can be displayed as a string of octets, but there is no parsing or creation of NDEF messages. However, the independent Python library ndeflib can be used for that purpose. It supports the most common NDEF types and can be used to convert a stream of bytes into an NDEF object that facilitates access to NDEF components.

Nfcpy features a list of tests for the different hardware and protocols. The user has access to a suite of examples for tag manipulation and reader/writer detection, as well as P2P communication. The examples do not contain code to fully emulate a T2T or T4T.

3.3 ST25 Library

The ST25 library is an NFC library written in C provided by ST Microelectronics [26]. The library is made for the STM32 MCU family by ST Microelectronics. It is split into two parts: an Radio-Frequency Abstraction Layer (RFAL), which supports various NFC protocols, and a specific implementation of the NFC driver for the variant of the ST25 chip. The NFC RFAL can be used with the ST25R3911, ST25R3916, ST25R95, ST25R200, and ST25R500 chip families. This includes the ST25R3916B described previously.

The RFAL contains code to read from and write to T1T, T2T, T4T, and T5T. It can also interact with proprietary ST25TB and ST25TV NFC tags. MIFARE tags cannot be interacted with unless they conform to an NFC tag specification, like MIFARE Ultralight tags. Tag emulation is not supported, despite the ST25R3916B offering tag emulation. It supports NFC-DEP, and it includes an NDEF parser for various NDEF record types.

The drivers for the different chips are all compatible with the RFAL. They contain functions to read and write registers on the chip and convenience functions to start data transmission or data reception. The driver uses a worker function that must be executed periodically to check the current state of the chip and to return an error if a timeout has occurred. This is a deviation from the IRQ-based mechanism used in the other libraries.

3.4 nRF Connect SDK

The nRF Connect SDK is a software development kit provided by Nordic Semiconductor [29]. It is based on Zephyr — an open-source operating system for embedded devices. It provides support for the nRF52840 NFC peripheral and the ST25R3911.

The SDK makes use of the nrfx set of standalone open-source drivers for the nRF52840 [30]. These drivers provide support for the nRF52840 NFC peripheral. Only T2T and T4T are supported in tag emulation mode, and the functionality is provided in pre-compiled libraries. This part remains closed source, but Nordic Semiconductor provides documentation on the API.

The ST25R3911 driver allows for communication with the device and implements the anti-collision loop for NFC-A, but it is not integrated with any high-level library. The repository provides samples to read a T2T or T4T with the ST25R3911.

TNEP polling is supported for the ST25R3911 and listening is supported for the nRF52840. The TNEP makes use of the closed-source T4T library. Samples are provided to make both devices work with TNEP.

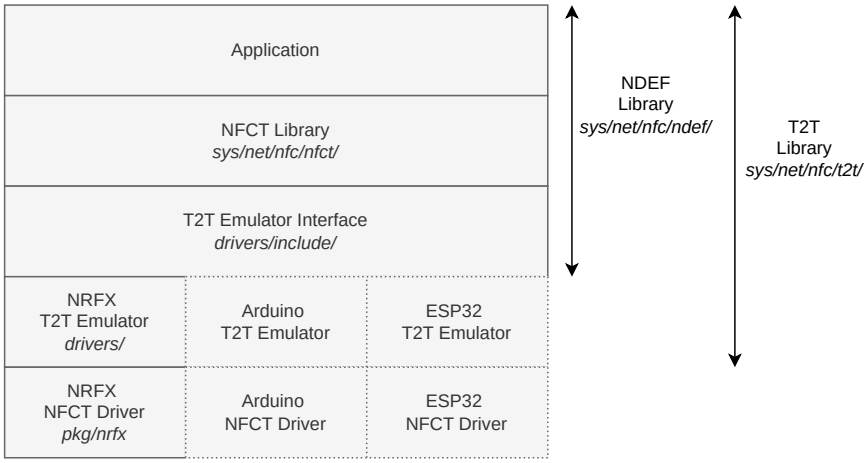

Figure 3.1: The current NFC stack in RIOT OS. The parts outlined by dots can potentially be added, but are not yet part of the NFC stack.

3.5 Current State of NFC in RIOT OS

RIOT OS features limited support for NFC. Notably, there is a driver for the PN532 that enables the user to access T4T and MIFARE Classic tags. T2T cannot be accessed. The driver provides functions to activate tags and read single blocks from a MIFARE Classic tag and send APDUs to T4T. However, the user has to know the correct interaction flow to read from and write to the respective tags, e.g., for a continuous read of a T4T, the user has to keep track of the current read offset. Therefore, there is no streamlined tag access provided to the user. Authentication of a MIFARE tag needs to be done manually as well by providing a key. Tag emulation is not supported at all.

Separate from this, a few pull requests are still pending that add the T2T library, NDEF library, and T2T emulation support on the nRF52840[1]. The full NFC stack can be seen in Figure 3.1. The T2T library allows the user to initialize a `T2T` struct. It provides functions to access and manipulate a tag. This includes block-wise read and write operations. The NDEF library provides a flexible way to create NDEF messages with a variable number of NDEF records. The supported record types are: text, URI, and MIME. NDEF messages can only be created by supplying the necessary information, but they cannot be decoded from a stream of bytes, as this is not necessary for tag emulation. NDEF records can be added up to a maximum amount defined at compile-time. They can be removed from an NDEF message.

The T2T emulation is based on the Nordic Semiconductor nrfx driver package [30], and it can only be used with the nRF52840 NFC peripheral. Emulation is built with some abstraction in mind: an additional compatibility layer is built on top of the nrfx driver named `nrfx_t2t_emulator`. The compatibility layer runs in its own thread and allows starting and stopping tag emulation. The functions defined inside it conform to a `t2t_emulator` interface. This extra layer finds its use in a high-level NFCT (NFC Tag) library. It provides easy-to-use functions for the user to start and stop emulation of a T2T given a T2T struct. It bridges the gap to the NDEF library by allowing the user to add NDEF messages with singular record types to the T2T.

[1] https://github.com/RIOT-OS/RIOT/pull/21240

NFC Library	Supported Chipsets
nfcpy	PN53X
libnfc	PN53X, PN71XX
ST25 Library	ST25 Series
nRF Connect SDK	nRF52840, ST25R3911B
RIOT OS	nRF52840, PN532

Table 3.1: This lists the supported NFC chipsets for each library.

NFC Library	Tags							P2P			
	T1T	T2T	T3T	T4T	T5T	MFC	MFU	NFC-DEP	LLCP	SNEP	TNEP
nfcpy	R/W	R/W	R/W/E	R/W			R/W	X	X	X	
libnfc	R/W	R/W	R/W			R/W	R/W	X			
ST25 Library	R/W	R/W		R/W	R/W			X			
nRF Connect SDK		E		E							X
RIOT OS		E									

Table 3.2: This showcases the supported tag types and P2P communication modes. The tag protocols are supported by the libraries in reader/writer mode when they are marked with a (R/W) and in tag emulation mode with an (E). The P2P communication modes are always supported for poller and listener, that is why they are simply highlighted with an (X).

3.6 Comparison of NFC Libraries

Comprehensive information on the libraries and RIOT OS can be found in the tables provided in this section. If a library only features example code for certain tag types or protocols that are strictly speaking not part of the library, it is not marked as supporting this feature. Table 3.1 provides an overview of the chipsets supported by each library. Table 3.2 shows the tags supported in reader and writer mode as well as the P2P communication protocols.

The libraries provide limited support for all NFC modes and tag types. The ST25 library only supports reader and writer modes and is limited to the ST25 chipsets. Libnfc only allows for full tag reads and writes. The nRF Connect SDK only supports tag emulation and the TNEP. Nfcpy has the richest support out of all of them and has an easy API to manipulate NDEF messages that can also be parsed by *ndeflib*, but it lacks easy tag emulation, just like the other libraries. RIOT's current NFC stack, as seen in Figure 3.1, is only suited for tag emulation and only for the nRF52840. A new NFC stack is needed that offers access to both reading and writing as well as tag emulation.

Both libnfc and nfcpy are unsuited for the IoT because the former makes use of dynamic memory allocation and the latter is a Python library unfit for constrained IoT environments. The nRF Connect SDK is written for the IoT OS Zephyr, but the tag libraries are only available pre-compiled. This creates the need for a novel IoT NFC library designed for compatibility with a myriad of NFC devices.

4 Design

This chapter will first consider possible interfaces between high-level libraries (T2T, T4T, MIFARE Classic, LLCP) and low-level drivers (nRF52840, PN532, PN7160, and ST25R3916B). After evaluating each of them, the interface deemed the most suitable for an NFC library of the constrained IoT is picked. The design of high-level libraries is outlined afterwards. The chapter ends with design considerations for low-level drivers.

4.1 Possible Interface Designs

This section will cover possible approaches to an interface between the high-level libraries and the low-level NFC device drivers. The approach must be flexible enough to accommodate the different device drivers, NFC technologies, and protocols. The abstraction layer will be evaluated in terms of scalability, i.e., the ease of adding new device drivers, the compatibility with the NFC standard, the resource consumption in light of constrained devices, the feature set exposed to high-level libraries, and the compatibility with the requirements of high-level libraries.

The architectures must consider the device's capabilities. Not all devices are capable of all NFC technologies and protocols. The supported features are sometimes supported only in hardware or software. The possibilities that are considered next are based on the position of the interface inside the NFC stack. This position governs the functionality that is implemented inside the hardware drivers or outside them. The possibilities are: a full design of the NFC stack in software in the form of a basic operations interface, a universal NCI interface for all NFC devices, and an approach that aims to find a balance between software and hardware implementations called the `nfcdev` interface. Each interface would provide function signatures and optional state that hardware drivers need to comply with.

4.1.1 Basic Operations Interface

The first approach is an interface that exposes only basic NFC operations. The approach can be seen in Figure 4.1. The basic operations exposed must be the reception and transmission of frames, as well as configuration options. Each driver would need to support these basic

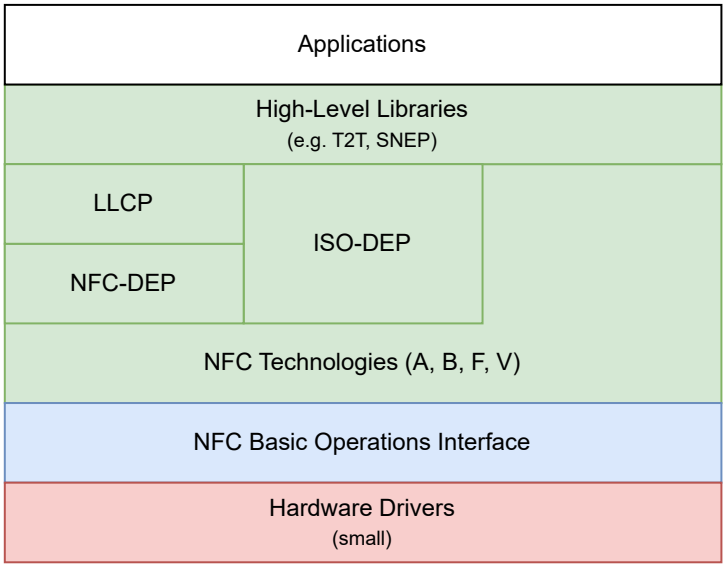

Figure 4.1: The NFC basic operations interface. Everything that is hardware-independent is highlighted in green and blue.

operations by defining functions to send and receive arbitrary NFC frames. The frame format, modulation, and configuration are different for every NFC technology. Each NFC technology also has a set of frame types that would also need to be supported by allowing the user to send isolated frames of any type.

Advantages

The advantage of such an approach is the added flexibility. Higher-level libraries would have access to send arbitrary frames and can be used with any hardware driver. The drivers themselves would be minimal with this approach, reducing code duplication.

Disadvantages

This approach makes minimal use of the NFC chipset's hardware features. By sending and receiving individual frames, there is no way to leverage the hardware's capabilities; therefore, all NFC technologies (A, B, F, V) and protocols like NFC-DEP and ISO-DEP need to be implemented in software. With software implementations of all NFC technology activities and protocols, additional drivers could be designed agnostically to any high-level protocols. This approach would obviously lead to slower execution times as more processing would need to be done on the host and less in hardware. Especially, the mandatory anti-collision activity at the start of every NFC communication requires an exchange of multiple commands that can be handled by most NFC devices in hardware. Each exchange would require sending data on the serial link to the host device, processing the command, and a response via the serial link back to the NFC device.

When all parts of the NFC stack are implemented in software, this means that the ROM size occupied by the program is bigger than if some of it had been left to the hardware. The stack size required would also need to be bigger to accommodate the extra calls necessary to traverse the added software layers. This increases the overall resource consumption, which is unfavorable on constrained devices.

ISO-DEP and NFC-DEP pose an additional challenge for such an approach: the Wait Time Extension (WTX) and Response Timeout Extension (RTOX) S-Blocks for both protocols need to be issued from the listener when more time is needed for the processing of an I-Block. In the same fashion, the poller needs to be ready to receive such a request for more time. This means that during the command flow and high-level processing of an I-Block's payload, the driver must run in a separate thread to monitor and handle the time extensions without blocking the rest of the application. This is something that does not have to be done when the hardware takes care of the protocol, as the asynchronous nature of this procedure is outsourced to the NFC chip or the peripheral. ISO-DEP and NFC-DEP must also support message chaining, and this would require extra implementation in software.

The MIFARE Classic tag protocol is integrated in hardware for the PN532 and PN7160. Without this hardware support, the libraries built on top must implement the authentication handshake in software, including the proprietary CRYPTO1 cipher to establish a secure communication channel with the tag. During the exchange, all commands must also be encrypted with the established shared secret. This makes a software implementation of MIFARE Classic tags infeasible.

A restriction of the PN7160 is that it cannot be used to send individual frames. The NCI operates with interfaces that take care of automatic framing. Short Frames, as defined in NFC-A, cannot be sent manually. The NCI is not designed to work with such low-level control. Therefore, NCI-capable hardware could not be supported with this interface design.

The anti-collision activity is strictly defined in the NFC specifications for every NFC technology. Explicit control over this activity has no added benefit. Except for type 1 tags, all tag specifications and P2P protocols comply with a technology-specific anti-collision activity.

The timing requirements are another problem for a software implementation of the various NFC activities. In the following section, the exact timing requirements will be scrutinized.

Timing Requirements For a library that aims to implement the poller as well as the listener part of NFC in software, timing is important. One has to consider the time it takes to assemble a frame, send it via a serial protocol to the NFC device, and also wait for a response. The poller part of the NFC standard is less problematic because the poller issues the request commands and the listener responds to them. For most of the NFC technologies, the timings between two consecutive request-response pairs are not defined, i.e., the poller can take an arbitrary amount of time to process the incoming response before sending the next request.

Timings for listeners are usually stricter. The timing for responses is strictly defined for all NFC technologies. In particular, for the anti-collision activities of each technology.

The transmission of the frame data from the host to an NFC chip like the PN532 takes time. If we assume a fast SPI connection of $5\,\mathrm{MHz}$, we get a data rate from the host device to the chip of roughly $5\,\mathrm{Mbit/s}$. This means the transmission of one byte takes

$$\frac{8\,\mathrm{bit}}{5\,\mathrm{Mbit/s}} = 1.6\,\mu\mathrm{s}$$

This does not take into account the implementation-specific time it takes for the host device to assemble the frame and the processing on the chip. For a software implementation of the listener, the time it takes for the listen frame to be transmitted to the host device needs to be added, too. This effectively leads to two serial transmissions before a response is transmitted to the poller.

The most time-critical part of the communication is the anti-collision activity, as strict timings need to be upheld. The derivation is outlined in paragraph 2.4.1. The listener needs to respond at a specific time called Frame Delay Time (FDT). For a standard bit rate of $106\,\mathrm{kbit/s}$ the FDT is $87\,\mu\mathrm{s}$. The NFC-A anti-collision activity uses commands of varying sizes, the biggest among them, and logically the one that takes the longest to transmit is 7 bytes long (SDD_REQ/SDD_RES). With an SPI speed of $5\,\mathrm{Mbit/s}$, the resulting serial transmission time is $11\,\mu\mathrm{s}$ for one trip. The full round-trip time is $22\,\mu\mathrm{s}$. Comparing the serial transmission time and the FDT leaves some spare time to process the frame as $FDT = 87\,\mu\mathrm{s} > 22\,\mu\mathrm{s}$, but the comparison doesn't take into account the time it takes to process and wirelessly transmit the frame. For a slower SPI speed of $500\,\mathrm{kHz}$ the round-trip time increases substantially $FDT = 87\,\mu\mathrm{s} < 90\,\mu\mathrm{s}$. The timing constraints can simply not be met with a serial connection.

Another problem is the exactness of the timing. The chip needs to take care of sending the data in a very tight time window around the FDT because software-based timers cannot reach nanosecond precision in the context of RIOT OS. For this reason, the NFC-A anti-collision activity can and must be done in hardware for the listener by providing the necessary information as required by NFC-A for the anti-collision activity in advance. Every NFC-A compatible hardware that supports the listen mode also offers a way to do the anti-collision procedure in hardware. A software-only approach cannot be used for the listen mode in NFC-A. The basic operations interface is entirely unfit for this case.

4.1.2 NCI Interface

The NCI interface aims to introduce the NFC Controller Interface (NCI) as a universal way to communicate with all underlying NFC hardware drivers. The NCI interface would then behave according to the NFC Forum NCI specification [25]. The design around the NCI interface can be seen in Figure 4.2. The NFC technologies and their activities, as well as the ISO-DEP and NFC-DEP, have been moved into the NCI interface. The LLCP has to be implemented in software, as the NCI does not handle the LLCP framing and operation.

Advantages

An advantage is that the NCI is a newer specification, and more recent chips like the PN71XX by NXP feature NCI support. In the future, more chip families might support this interface, which means that driver development becomes less tedious as each new driver would only need to be slightly changed to account for vendor-specific changes. New chip families might define

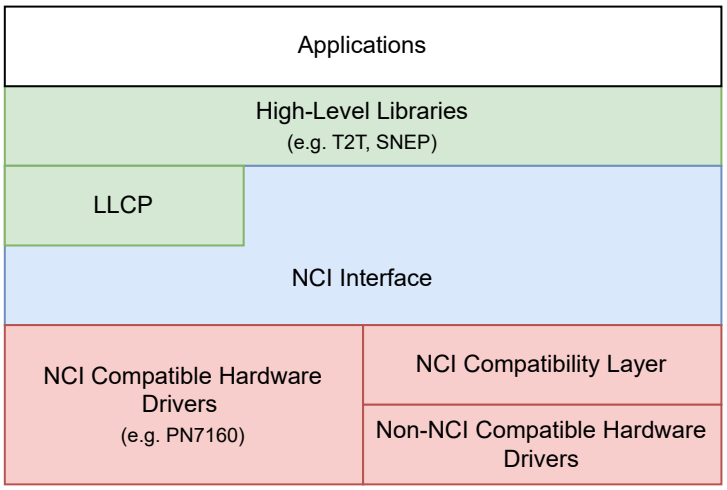

Figure 4.2: The NCI interface.

custom RF interfaces, but the standard set of NCI RF interfaces is guaranteed to be included in any NCI-compatible chip.

It should expose the NCI core interface to configure the NFCC and a way to interact with the RF interface to communicate with other devices. The NCI specification defines additional interfaces for protocols built on top of basic RF functionality, like the ISO-DEP or NFC-DEP interfaces. This is a considerable improvement over the basic operations interface, as the high-level library could simply leverage these interfaces to send payload to the hardware, and the NCI would take care of ISO-DEP and NFC-DEP framing and protocol operations.

For NCI-compatible devices, the NCI state machine would not be implemented in the driver but managed by the hardware. The design approach would mean that every NFC driver needs to expose the same set of functions, where each function corresponds to an NCI command. The response would be returned by each function to the library.

Disadvantages

The biggest drawback is that the NCI is only supported on a subset of all NFC devices, as visible in Table 2.8. Only the newer PN71XX family supports the NCI, and older devices have their own interface. The ST25R3916B is an RF frontend and has no command set to interface with the host at all; neither does the nRF52840 support the NCI. These three NFC devices would need an additional compatibility layer that translates NCI commands into hardware-specific commands or register manipulations. The NCI compatibility layer would have to be written for every NFC device, as there simply is not a one-to-one translation of every NCI command to every non-NCI firmware command. The NCI compatibility layer must necessarily keep track of the NCI state at every given time. This places a heavier burden on the development of new drivers as the developer would need to correctly identify the device's internal state and translate it to a corresponding NCI state and configuration.

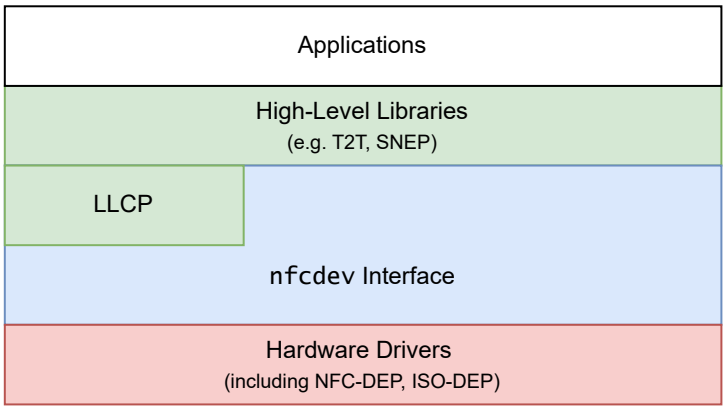

Figure 4.3: The `nfcdev` interface.

The interface itself would be made up of many functions to cover all configuration options. A balance could be struck by combining multiple NCI commands into new procedures. This would decrease the interface's complexity.

A problem with this design approach is that developers implementing or verifying high-level libraries for NFC would need to understand a greater number of API entry points required by the NCI and the valid commands for each state, as not every NCI command can be issued in every state of the NCI state machine. For this, the NCI specification has to be acquired from the NFC Forum, and it is not openly available elsewhere.

4.1.3 `nfcdev` Interface

The `nfcdev` interface is an attempt to find a balance between a full software implementation of all protocols and technologies of the NFC stack and a full hardware implementation that requires hardware support for all features. The interface can be seen in Figure 4.3. Similar to the NCI, the ISO-DEP and NFC-DEP are handled by the NFC hardware driver. This means that the implementation of these protocols has to be device-specific. The LLCP remains above the interface, as none of the NFC devices described earlier can handle the LLCP in hardware.

The `nfcdev` interface would expose functions to change the device into a poller or listener state, as well as functions for data exchange. These functions can roughly be described as `poll` and `listen` functions for the state changes. The functions for data exchange must then frame the data in the correct format internally for whatever protocol is currently in use. If an NFC chip takes care of any protocol, it wouldn't have to be done, and the framing is outsourced to the NFC chip.

High-level libraries could query for nearby NFC tags by putting the device into a poll state, or they could start tag emulation by putting the device into a listen state. After these functions return with success, the actual data exchange takes place.

Advantages

Unlike the other two interfaces, this design is flexible to work with all the hardware mentioned earlier. If a protocol is implemented in software because it is not natively supported on an NFC chip, it can still be used by multiple NFC devices.

The NCI features a larger set of commands than the `nfcdev` interface, but a subset of this command set can be used to emulate NFC functionality. The NCI can be used for the PN7160 and later driver implementations of other NCI-compatible NFC drivers.

This `nfcdev` interface is less complex than the NCI or basic operations interface presented earlier. The interface can be used without understanding the entire NFC protocol stack. This makes it easier to directly use the interface in new high-level library implementations. Simple state tracking is required to exchange data.

New drivers can be added by defining the interface's functions. The smaller interfaces facilitate the integration of new drivers. Developers only need to expose a smaller subset of functions, unlike the other two interfaces, which expose a greater number of functions. Most of the protocol functionality is integrated into the driver.

Disadvantages

The implementation burden is higher on those devices that do not feature a high-level command set. This increases the implementation overhead for the drivers of the nRF52840 and the ST25R3916B, but it also means that the drivers for the PN532 and the PN7160 are easier to implement. The smaller interface also means that the driver must make some assumptions about certain device-specific configuration options that simply cannot be exposed to the high-level libraries.

Conclusion

This approach is the best suited for an NFC library among all three outlined here. It is easily expandable and less complex in its function set than the other two. Despite this simpler function set, all functionality can still be realized with this interface. Because of these reasons, it was chosen as the interface for this NFC library.

4.2 High-Level Libraries

The high-level libraries are built on top of the `nfcdev` interface. They can be tag libraries for reading, writing, and emulation. The libraries for tags are split into a reader/writer part and a tag emulation part, which use the `nfcdev` interface. Certain functions of each library only manipulate a local tag representation and are therefore independent of the interface. These functions include the creation of a tag in preparation for tag emulation or the printing of tag information. The high-level libraries make up the layer that applications use to interface with

the NFC stack. The developer of such applications needs to understand the `nfcdev` interface, but not the underlying NFC technologies.

Tag interaction is almost exclusively done in the context of NDEF messages, and any tag library that will be implemented offers the option to directly extract or overwrite the embedded NDEF message. The unified NDEF access makes it possible to develop a wrapper around various NFC tag libraries to edit the NDEF message without knowing the tag type that the application will operate on. Despite this, if a user wants to access the raw tag data — including meta-information — this functionality should also be exposed, even if less practical, because the user might want to access the tag for debugging purposes.

SNEP and LLCP are only used in P2P mode, and the former necessitates the latter. They can make use of the `nfcdev` interface just like the tag libraries.

4.3 Low-Level Drivers

Each NFC device has its own driver, and each driver needs to implement the interface's functions and keep the current driver state and the activated protocols. A low-level driver should allow for communication over multiple serial protocols if the driver supports them. The exact implementation will naturally differ greatly based on the NFC device's capabilities.

5 Implementation

The chapter starts with a look at the implementation details of the `nfcdev` interface. It will then cover all high-level NFC tags and P2P libraries, and it ends with the individual NFC device drivers. The structure of the entire NFC library with all of its components can be seen in Figure 5.1. The code for the library can be found on a fork of the RIOT project inside the `nfc-library` branch[1].

5.1 `nfcdev` Interface

The `nfcdev` interface is defined in its own header file `nfcdev.h`. The `nfcdev_t` type is a struct that has a pointer to the underlying NFC driver, a pointer to an `nfcdev_ops_t` type struct, and an `nfc_dev_state_t` type enum that stores the current state of the `nfcdev`. The old `t2t_emulator` interface was replaced by the more generic `nfcdev` interface.

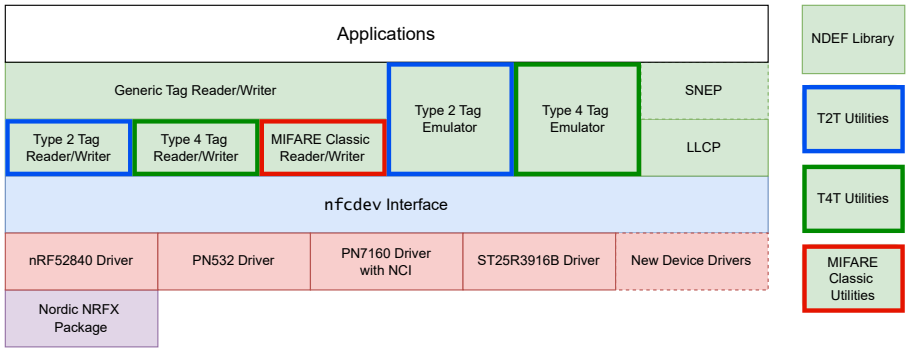

Figure 5.1: The red parts of the NFC library are hardware specific. The Nordic Semiconductor NRFX package [30] is also hardware-specific and included as an external package. The green parts are hardware-independent, just like the blue `nfcdev` interface. The free-floating utilities on the right are used by the multiple parts outlined by the same border, and are also seen in use on the highest application layer. The utilities only contain functions to operate on local tag structs. The SNEP has dashed lines because it is not part of the implementation, but can be added on top of the LLCP. Additional device drivers can also be added.

[1] https://github.com/N11cc00/RIOT/commit/99b97740fff24c09c9fb43cddd86e6d214e28f37

The state needs to be modified by the driver. With this, the driver can ensure that only the correct functions are called in the current state. The states are `NFCDEV_STATE_UNINITIALIZED`, `NFCDEV_STATE_IDLE`, `NFCDEV_STATE_POLLING` and `NFCDEV_STATE_LISTENING`. If a driver needs to implement a custom state for the specific device, the NFC driver struct inside the `nfcdev` can be defined to hold variables for internal state-tracking.

The interface has function pointers in an `nfcdev_ops_t` type that defines a struct that has all the necessary operations for the NFC standard. Every function takes a pointer to the `nfcdev`. The functions are `init`, `poll` and `listen` and variants of these functions for different NFC technologies, `target_send_data`, `target_receive_data`, `initiator_exchange_data` and `mifare_classic_authenticate`.

The functions in this type are an `init` function to initialize the underlying driver. Each driver has its own configuration that needs to be passed. It must be called before using any of the other functions defined in the interface. This transitions the device into the `NFCDEV_STATE_IDLE` state.

5.1.1 Poll and Listen Functions

The `poll` function is defined for every NFC technology type, as the activities are different for each of them. Therefore, there are four functions: `poll_a`, `poll_b`, `poll_f`, and `poll_v`. If a listener is found, its technology-specific data is returned in an `nfc_listener_config_t` struct to the caller. Four variants are defined for each of the four technology types. Depending on the technology, it contains the SENS_RES and SEL_RES for NFC-A; for NFC-B, it is the SENSB_RES; NFC-F uses the SENSF_RES; and NFC-V uses the INVENTORY_RES. Each NFC technology also has a unique ID, e.g., an NFCID1 for NFC-A. This NFCID1 is stored alongside its length in the struct. High-level libraries can use this information to infer the capabilities of the listener.

If no listener device is found inside a timeout window, the function returns an `NFC_ERR_POLL_NO_TARGET`. After this failure, the function can be called again to poll for a target. If the function has any kind of high-level protocol available, the driver is expected to put the device into the activated state. For ISO-DEP, this means sending the RATS and receiving the ATS command; for NFC-DEP, driver-level exchange must include the ATR_REQ and ATR_RES. The idea is that a library calls this function, and if the function succeeds with a listener found, other function calls can be made to exchange data.

The universal `poll` function works independently of any particular NFC technology. This function polls for all supported NFC technologies of the device and returns a union of all previously covered `nfc_listener_config_t` structs. The added benefit is that the high-level library does not need to know the exact technology used by the listener presented to the NFC poller. The drawback might be a higher execution time compared to the explicit poll functions, depending on how the NFC driver implements this function. Some devices can immediately poll for multiple technologies at once, while others need to be switched between configurations until all technologies have been tested.

An important case is the potential presence of multiple listeners in the RF field. If there are multiple listeners in the field, then only one is activated and can be interacted with. This is standard NFC behavior. The driver or the NFC chip decides which tag to activate first. The API

still makes a compromise because the user is not able to select which of the listener devices to activate. A tag has to be deactivated first before other tags can be activated. This is acceptable, as in most real-world applications, only one tag is in the field of the poller, and the other tags can still be accessed after the first one has been put to sleep.

A minor drawback of the universal `poll` function is that specific activation of a tag is not supported because the poller cannot be configured, i.e., there is no `nfc_poller_config_t`. Such a configuration could include the NFCID of a listener to activate only a listener with the same NFCID. The problem is the expected behavior of `poll` to always activate the first target it finds. The poller would need to issue a poll at least twice: the first time to query all tags in the RF field and the second time to activate one tag. If the poll function were to be made to return multiple listener configurations, it would return them in a buffer. It was decided to return only the currently activated tag.

Analogously to the `poll` functions, there are `listen` functions defined for every NFC technology. They behave differently, as they block until a poller has been found or a timeout has been reached. They require a configuration passed as a `nfc_listener_config_t` struct. Unlike the `poll` functions, there is no universal `listen` function, as a tag cannot be emulated without knowing the technology it should use.

5.1.2 P2P Functions

An additional pair of `poll` and `listen` functions is the polling and listening for NFC-DEP. These functions are solely used to put the NFC chip into the NFC-DEP P2P mode. Due to the variable transmission speeds, the functions take a baudrate argument to configure the chip to only operate at a specific speed.

5.1.3 Data Exchange Functions

The initiator can send and receive data with the `initiator_exchange_data`. The function takes a buffer to send and a buffer to receive data. The outgoing data is `const`, so it is not modified inside the function. The function sends and receives data. Two length values must be passed along: the first one for the length of the outgoing data buffer and the second one for the length of the reception buffer. This allows the driver to check for potential overflows if the incoming data exceeds this limit. Internally, the data is copied into the transmit buffer and copied out of it when a response has arrived.

The `target_send_data` and `target_receive_data` work similarly for the target side. They should be called only after the NFC device has been activated after a successful `listen`. Unlike the equivalent function for the initiator, the sending and receiving of data are split into distinct functions. This matches the request-response model of NFC because the initiator sends data and waits for a response by the listener, but when the listener receives data, it must process the data first returned by the `target_receive_data` function before sending a response with the `target_send_data` function, which can be called when data is ready.

5.1.4 Proprietary Functions

The only vendor-specific command is the `mifare_classic_authenticate` function. As stated earlier, the MIFARE Classic authentication handshake cannot be implemented in software, and the firmware of each NFC chip expects different parameters to initiate the handshake. For this reason, it has its own function inside the interface. The function takes a block number inside the sector it needs to unlock, the NFCID1 of the tag, a boolean to use key A or key B, and the 6-byte key itself. After this function call, the driver is in a state where it can read or write the unlocked sector. If the authentication fails, the function returns an `NFC_ERR_AUTH`. The caller can try again with a different key or block number.

5.2 High-Level Libraries

The high-level libraries can be classified into the tag libraries and the P2P libraries. The tag libraries all share some common API to read and write NDEF messages onto a tag. The tag libraries are a T2T library, a T4T library, a MIFARE Classic library, and a generic tag reader/writer wrapper for the other tag libraries. Every tag library is split into a utility part that only operates on a tag struct, a reader/writer part, and a part for tag emulation — if supported by the tag type. The P2P libraries are limited to the LLCP.

All reader and writer parts provide a `nfc_tag_rw_read_ndef` and `nfc_tag_rw_write_ndef` to only operate on the NDEF of a tag. For emulation, there is only the `nfc_tag_emulator_start`. This function blocks until the emulation stops; the user needs to create their own thread if asynchronous behavior is desired.

5.2.1 Changes to the NDEF Library

The NDEF library was modified to allow for the creation of an NDEF message from a buffer. The `ndef_from_buffer` parses the buffer inside the NDEF message to find the count and the start of each NDEF record.

5.2.2 T2T Library

The T2T library is made up of functions to operate on T2T structs called utilities, a tag reader/writer, and a tag emulator. Functions as part of the utilities for local tag manipulation are covered first.

The T2T struct is defined as a tightly packed struct of bytes to match the T2T layout. The struct is segmented into the internal bytes, the lock bytes, the CC, and a data array. This means that the size of the data array must be known at compile time. The size is set to a default value of 64 bytes, which reflects the T2T static memory size configuration. But the value can be changed by setting the config option `CONFIG_NFC_T2T_MEMORY_SIZE` to any value up to 2048 bytes.

A tag can be locked to be read-only. The NDEF message and the tag UID can be extracted from the T2T. A T2T can be initialized with an NDEF message. The user can also print the T2T in a

block-wise format. The two functions `t2t_write_block` and `t2t_read_blocks` take a block and sector number to read from. Functions to create and parse TLVs are included in a separate `tlv.c` file. They are only needed for T2Ts, but they can be reused for proprietary tags that also use TLVs.

The T2T reader/writer has its own struct just like all readers and writers. It keeps a pointer to the `nfcdev` currently in use. This reader/writer struct is then passed around inside the function. The two NDEF read/write convenience functions make an initial call to the `poll_a` function, as T2Ts must be using NFC-A. The CC is extracted to infer the tag's size and read/write access. This size is checked before every operation to ensure the tag reader/writer never reads or writes beyond the tag's memory boundary. The first TLV inside the tag must be the NDEF TLV. The reader/writer can be used on standard T2Ts but also on T2T-compliant proprietary tags like the MIFARE Ultralight tags without utilizing the proprietary commands.

In the case of reading, this TLV is parsed until the end of the NDEF message has been reached with consecutive read operations. The sector select command is not implemented, so the maximum tag size that can be read is 1024 bytes. T2Ts are rarely bigger than this size, unlike T4Ts. The MIFARE Ultralight tag used for testing later has a size of 540 bytes. If the NDEF message has been read completely, it gets parsed, and the function returns. For NDEF writing, the procedure is similar to NDEF reading, but the correct TLV is written to the tag first, and then the NDEF message is written block-wise. After the NDEF write operation, a terminator TLV is placed. The user also has the option to read the entire tag, including the reserved bytes at the start of each tag, into a T2T struct. The procedure is less complex as no extra NDEF parsing has to be done.

At the end of these read or write functions, the tag is put into a sleep state with a SLP_REQ. In the sleep state, the tag cannot be polled and activated again unless it is taken out of the RF field. This means an additional call to one of the read or write functions activates other potential tags in the RF field.

For tag emulation, the listener blocks until a poller activates the tag or a timeout, which is defined at compile time, happens. The tag then acts fully T2T-compliant and supports all T2T commands. This includes the sector select command that is not supported by the reader/writer. The emulation ends when a timeout occurs or the tag receives a SLP_REQ from the poller.

5.2.3 T4T Library

The T4T struct is defined in contrast to the T2T not as a struct of continuous bytes, but it contains meta-information on the tag's state that resembles a file system. The T4T has a CC file as a fixed struct of 15 bytes (the CC file cannot have a different length) and a pointer to a byte array that represents the NDEF file. The `CC` file and the T4T struct require the compiler extension `__attribute__(packed)` because of 8 and 16-bit unsigned integers mixed together inside the struct. This allows the structs to be copied with `memcpy`. The T4T keeps track of the selected file and the maximum NDEF file size. A T4T can be initialized, and the user only needs to supply the maximum command APDU size that is accepted by the tag. An NDEF message can also be added. All the required APDUs for T4T activation are saved in the `t4t.h` file as they must exactly match. The T4T library includes an APDU library for verification and extraction of data.

The T4T library expects reassembly of multiple ISO-DEP I-Blocks to happen in the driver so that information is directly received in full APDUs without fragmentation.

The reader and writer only implement the read and write NDEF functions. For the case of the T4T R/W, the struct tracks an extra variable that tells the T4T library whether the tag has already been selected by a previous call to `poll`. If this is the case, the initial `poll` is skipped. The reason for this extra logic is that T4Ts, unlike T2Ts, cannot be polled again once they are activated. If this is the first function call, they poll for NFC-A and NFC-B. Both query the CC file of the tag first to determine the maximum command APDU size. The maximum APDU size is the minimum of the buffer size for sending and the maximum command APDU. The requested data of read APDUs should never exceed this buffer size either.

The T4T emulation starts with a listen for NFC-A, and after a successful connection, the T4T responds to all possible APDUs according to the operation specification. It ends when the data exchange function returns because of a timeout or a deselection command.

5.2.4 MIFARE Classic Library

The MIFARE Classic tag is represented as a struct that contains the two possible size options for MIFARE Classic tags as a union. Either size is represented as a tightly packed array of bytes that is itself segmented into sectors and blocks. This allows for seamless access to the tag by treating it as an array of bytes. Initialization of tags is not supported, as tag emulation is not supported.

The MIFARE reader and writer need to make sure to constantly authenticate the tag when reading and writing for every new sector that is accessed. The authentication happens with a convenience function inside the reader and writer. For MIFARE Classic tags, the keys are predefined, as they are always the same for unaltered MIFARE Classic tags. The user can supply keys by defining them in the `make`-build system if a tag's keys have been changed. A drawback is that keys are not selected on a per-sector basis. The implementation of a custom key selector will be implemented at a later point but the function for authentication with arbitrary keys already exists internally.

The tag's NDEF message can be read by parsing the NDEF TLV similar to T2Ts. In the same way, it can also be modified. While reading or writing, the trailer blocks have to be skipped. If the user wants to extract the keys, the entire tag can also be dumped under the assumption that the tag allows for trailer block access. Due to the proprietary nature of the MFC protocol, tag emulation is not implemented.

5.2.5 Generic Tag Reader/Writer

The generic tag reader and writer is a wrapper around the other reader and writer parts of the other tag libraries. It allows for reading and writing whole NDEF messages to T2Ts, T4Ts, MFC tags, and by extension, MIFARE Ultralight tags. It first does a general `poll` for all technologies. If a listener is found, its technology is determined by inspecting the returned `nfc_listener_config_t` struct. The technology-specific data is then analyzed to infer the tag

type and vendor. With this information, the correct reader/writer can be instantiated, and the NDEF message is accessed.

5.2.6 LLCP Library

The implementation of the LLCP is split into an LLCP controller and an LLCP socket. It expects the underlying driver to handle the NFC-DEP. The controller runs in its own thread and has an array of LLCP sockets. The sockets themselves have two thread-safe ring buffers — one for receiving, one for sending — taken from the RIOT `tsrb.h` header that guarantee safe access from two different threads. Additionally, each socket has a SSAP and DSAP. The idea is that an LLCP controller manages a variable number of LLCP sockets that are uniquely identified by their SSAP-DSAP pair.

A socket must be initialized first by specifying the SSAP and DSAP and the socket mode. As of now, the only socket mode supported is the connectionless socket mode. A socket has two functions that are used to send and receive data. The functions read information from the ring buffers. The data inside the ring buffers is actual payload data that is prefixed with a single length byte that tells the functions how much data to read. In the same way, data written to the ring buffer is prefixed with a length byte.

After initialization, the LLCP controller starts a separate thread that initializes a message queue for inter-thread communication and then constantly loops over its array of sockets. Sockets can be added or removed at any time by calling the add or remove socket functions. At the start of each iteration, the message queue is checked for a new message. The only message type currently defined is a message that zombifies the current thread. This message is sent inside the stop function that sends the message and kills the thread upon zombification.

To send data, the send buffer of each socket is read from, and the data is packaged into a UI PDU. It is then sent and received with `initiator_exchange_data` or the `target_send_data`/`target_receive_data` functions. The specific function invocation depends on whether the `nfcdev` acts as an initiator or as a target. If there is no data available in the buffer, the SYMM PDU must be sent. The function performs a short sleep that is shorter than the LTO. Parameter exchanges with PAX PDUs are not supported. The LLCP can be used without implementing the SNEP on top.

5.3 NFC Hardware Drivers

There are four NFC hardware drivers written for the new NFC stack. One written for each of the NFC chips or peripherals: nRF52840, PN532, PN7160, and ST25R3916B. The following sections will cover all of them in greater detail and will highlight their current capabilities. All drivers comply with the `nfcdev` interface. An overview of the supported tag types, protocols, and NFC technologies for each of the driver implementations can be found in Table 5.1.

All hardware drivers feature a timeout mechanism that functions by locking a mutex at the start of every communication with the chip and starting a timer. If the timer expires before a communication has been completed, the function returns with a timeout error. The default timeout is set to 2 seconds for serial transmissions.

Driver	Tags			P2P		Poll				Listen			
	T2T	T4T	MFC	NFC-DEP	LLCP	NFC-A	NFC-B	NFC-F	NFC-V	NFC-A	NFC-B	NFC-F	NFC-V
nRF52840	E									X			
PN532	R/W/E	R/W/E	R/W	X	X	X	X	X		X			
PN7160	R/W	R/W				X	X						
ST25R3916B	R/W	R/W				X							

Table 5.1: This showcases the supported tag types and P2P communication modes for the implementation of each NFC driver. The tag protocols are supported in reader/writer mode when they are marked with (`R/W`) and in tag emulation mode with an (`E`). The P2P communication modes are highlighted with an (`X`). Poll and Listen columns show support for specific NFC technologies with an (`X`).

5.3.1 nRF52840 Driver

The nRF52840 driver was modified from its previous version inside the existing RIOT OS NFC implementation. The old stack was narrowly built around T2T emulation functionality. It can be seen in Figure 3.1. The nrfx nfct driver by Nordic Semiconductor is still being used [30], but the `nrfx_t2t_emulator` was renamed into `nrfx_nfct_dev`. The driver is different from the others because the NFC peripheral on the nRF52840 is limited to one. That's why the header file defines a universal `nfcdev` struct that should be used for all related API calls.

The implementation of the driver was changed to no longer run in its own thread. Asynchronous tag emulation has to be implemented by high-level libraries. As of now, the driver only supports T2T emulation because the ISO-DEP protocol has not yet been implemented for this driver, and the hardware does not handle ISO-DEP automatically.

The driver has a few drawbacks because it relies on the nrfx package provided by Nordic Semiconductor. The driver inside the nrfx package that is used for the implementation in RIOT OS has a few compilation warnings because of unused variables and misaligned casts, and the ISR stack size has to be increased regardless of one's own implementation. On top of that, the nrfx package triggers a `field lost` event even when there is a capable reader/writer in the antenna's range. This slows the operation of the driver down. The nrfx package also occupies an IRQ that is used for internal timing functionality. This IRQ is also used for the `ZTIMER_USEC` in RIOT OS on the nRF52840; therefore, the microsecond timer cannot be used alongside the nrfx package unless a patch is applied to use a different IRQ for the nrfx package.

5.3.2 PN532 Driver

RIOT already provides a driver for the PN532 chip. The driver was heavily modified to comply with the `nfcdev`. It now supports polling not only in NFC-A but also in NFC-B and NFC-F, as well as listening in NFC-A.

The driver allows for the arbitrary sending and receiving of data. Previously, the driver only allowed receiving a fixed size of data; if the size of the received frame was not known in advance, extra data would be sent from the PN532 to the host. This is inflexible because the NFC standard has no fixed length for common data frames, such as NFC-A standard frames. With proper

parsing, the driver can infer the packet length by only reading the first few bytes and extracting the packet length of the PN532's packet format. This speeds up the serial link because only the required data is transmitted. The implementation supports SPI and I2C but not UART, because RIOT's UART driver works with a callback. Unlike the SPI and I2C drivers, which directly return data. The PN532 driver requires the PN532's IRQ and reset pins.

The ISO-DEP and NFC-DEP operations are done on the hardware by issuing an internal command to configure the driver for automatic ISO-DEP and NFC-DEP handling, and the only thing that the driver keeps track of is an `iso_dep` boolean that tells other functions whether ISO-DEP is currently in use. The PN532 requires different commands for data exchange. If the PN532 is configured as a listener, it uses `tg_get_data` and `tg_set_data` functions for ISO-DEP and NFC-DEP, and otherwise it calls the `tg_get_initiator_command` and `tg_response_to_initiator`. The former two functions handle all necessary protocol operations like chaining and transmission errors. If the PN532 is used as a poller, it always uses `in_data_exchange`. The function handles basic data exchange as well as NFC-DEP and ISO-DEP.

When the PN532 gets activated by a poller in listen mode, the PN532 returns the first frame after the device has been activated. This command can be a tag-specific protocol. To ensure normal protocol operation, this command must be buffered and returned when the `target_receive_data` is called. The other devices don't need such a workaround.

5.3.3 PN7160 Driver

The PN7160 driver contains an implementation of the NCI that can be reused by others. Most of the NCI definitions are listed in the `nci.h` header. The required `nfcdev` functions call the required NCI functions implemented in the driver.

For polling with NFC-A, the standard protocols are already mapped to the correct interfaces, i.e., ISO-DEP to the ISO-DEP interface. The only exception is the special TAG-CMD interface as outlined in subsection 2.9.3 required for MIFARE Classic tags. The TAG-CMD interface is mapped to the MIFARE Classic protocol. The driver keeps track of whether it is currently targeting a MIFARE Classic tag. If this is the case, the TAG-CMD's special header must be prepended when sending data or removed for high-level libraries when receiving data. The MIFARE Classic authentication as part of the TAG-CMD interface is done without an NFCID1, unlike the PN532, so it is discarded for this purpose. Extra care has to be taken because the PN7160 does not execute the MIFARE Classic write operation in one operation, unlike the PN532. The exchange function checks for the current operation, and if it's a write operation, it follows the MFC two-part write procedure.

The listening mode is implemented in the driver by configuring the NCI's routing table to route incoming ISO-DEP and NFC-A data to the host. The current interface is deactivated instead of returning information about the poller when it detects an RF field in the proximity. This makes the listen mode unusable with the PN7160. The cause could be an issue with the breakout board used for testing or a faulty configuration in the driver implementation[2].

[2] https://community.nxp.com/t5/NFC/PN7160-card-mode-RF-DEACTIVATE-NTF-after-discovery-notification/m-p/1902137

5.3.4 ST25R3916B Driver

The ST25R3916B driver includes more functionality in software than the other drivers because it only acts as an RF frontend. The software implementation can be reused for other drivers in the future, which only offer low-level access. Communication with the device is supported via SPI. Just like the PN532 and the PN7160, it uses an IRQ to inform the host when an event occurred, but unlike the other two devices, the cause of the event has to be read from a special IRQ register.

The poll mode for NFC-A is implemented in software. It is invoked when calling the `poll_a` function. This involves a simplified implementation of the anti-collision activity for NFC-A as described in paragraph 2.4.1. The simplified anti-collision loop works with one tag in the RF field. The ST25 sends out a SENS_REQ and waits for a SENS_RES. The SENS_RES contains information about the tag's vendor and is saved to the configuration struct. After all stages of the anti-collision loop involving the SDD_REQ/SDD_RES and SEL_REQ/SDD_RES pairs, the last SEL_RES is saved, too. The SDD_REQ is always filled with the entire NFCID1 for the current cascade level. The anti-collision loop concludes. The anti-collision loop cannot handle IRQs triggered by bit collisions at the moment, but they can only occur with multiple tags in the RF field, so single-tag operation is guaranteed to work. Handling of bit collision still needs to be implemented.

The driver needs to take care of technology-specific CRC verification. This is done according to the CRC calculation taken from ISO/IEC 14443 [14]. For this reason, the driver keeps track of the active NFC technology.

The code for the listen mode is part of the driver, but the board never triggers an IRQ when an RF field is detected. The listen mode is therefore not functional inside the driver.

To make the ST25R3916B compatible with T4Ts, it requires an implementation of the ISO-DEP. The ISO-DEP is implemented in a simplified form. The first step is to handle the RATS/ATS exchange. This exchange establishes the maximum frame size in both directions of the exchange. The maximum frame size is then estimated to be the minimum of the poller and listener maximum frame sizes. The ST25R3916B sets its maximum frame size to never exceed the size of the reception buffer, so a single ISO-DEP block can fit in the buffer. An important note here is that the maximum APDU size sent by the T4T library should not exceed the maximum frame size, as this would cause message chaining, and this feature is not supported in this implementation. As stated in the design chapter, the ISO-DEP would require its own thread for WTX. The extra thread is omitted here because in practice the T4Ts have minimal processing time and never request a WTX. With these facts established, there is no need for R-Blocks, and the S-Block is limited to tag deselection.

5.4 Example Usage

This section dissects one example of generic tag reading for the PN532. This snippet requires the `generic_rw` and `pn532_spi` RIOT modules. The code can be seen in Listing 1. For the sake of brevity, the code does not handle errors by the invoked functions. The top of the file includes the PN532 driver and the generic tag R/W header. Outside the `main` function, we define the NDEF message and NDEF buffer. Both should be kept outside the stack to save stack size.

```
#include "pn532.h"
#include "net/nfc/generic/generic_rw.h"

ndef_t ndef_msg;
uint8_t ndef_buffer[512];

pn532_config_t config = {
    .params = {
        .spi = SPI_DEV(0),
        .nss = GPIO_PIN(1, 1),
        .reset = GPIO_PIN(1, 2),
        .irq = GPIO_PIN(1, 3)
    },
    .mode = PN532_SPI
};

pn532_t pn532_dev;

nfcdev_t nfcdev = {
    .dev = &pn532_dev,
    .ops = &pn532_ops,
};

nfc_generic_rw_t rw;

int main(void) {
    nfcdev.ops->init(&nfcdev, &config);
    ndef_init(&ndef_msg, ndef_buffer, sizeof(ndef_buffer));
    nfc_generic_rw_read_ndef(&rw, &ndef_msg, &nfcdev);
    ndef_pretty_print(&ndef_msg);
    nfc_generic_rw_write_ndef(&rw, &ndef_msg, &nfcdev);
    return 0;
}
```

Listing 1: Example code to read a generic NFC tag with a PN532

The PN532's configuration and the PN532 device follow right after that. The `nfcdev` requires a pointer to the actual device driver that will be used and the device-specific `ops` struct. The reader/writer is also defined. This definition can be put into `board.h` header to use them for all devices. In the `main` function, we only ever use the `nfcdev`. The device needs to be initialized first with the `init` function. It is important to call `ndef_init` on the `ndef_msg` struct before use so that the internal state of the NDEF message is initialized. The `nfc_generic_rw_read_ndef` function returns the NDEF message. If there is no tag, the function returns with an error. If this function call succeeds, the NDEF message can be modified or written to another tag. In this example, the NDEF message is printed with the `ndef_pretty_print` function and then written to another tag with the `nfc_generic_rw_write_ndef` function.

Analogously to this example, the `nfc_generic_rw_write_ndef` function can be called to write an NDEF message instead of reading it. The NDEF message requires an NDEF record that can be added with one of the helper functions inside the NDEF library.

6 Evaluation

This chapter evaluates the new NFC library for RIOT OS by looking at the memory usage, the timing, and the power usage for plausible test cases. The hardware tags used for testing are a MIFARE Ultralight tag with a capacity of 540 bytes as a stand-in for an NFC T2T, a MIFARE DESFire EV3 tag with a memory area of 4048 bytes that acts as an NFC T4T, and a MIFARE Classic 1K tag for the MIFARE Classic tag, abbreviated as MFC. Unless stated otherwise, the tests were done on the nRF52840 development kit by Nordic Semiconductor using the SPI interface at a frequency of $1\,\mathrm{MHz}$.

All tests and results, including the Python scripts used for evaluation, can be found in a public GitHub repository[1]. The tests were done with link time optimization enabled and without debug options to increase performance and decrease resource consumption. Debug options were disabled by defining the `NDEBUG` variable and setting `DEVELHELP` to 0 in each Makefile. Logging was disabled with the `LOG_LEVEL` variable set to `LOG_NONE`.

6.1 Memory Size

This section looks at the ROM and RAM usage of the library as well as the maximum stack size.

6.1.1 ROM and RAM Usage

This section analyzes the ROM and RAM size for every NFC device and library in read/write and emulator configurations. The ROM size is composed of all data stored in the `text` and `data` sections. The Figure 6.1 depicts the reader/writer ROM size. The comparison for tag emulation is in Figure 6.2. None of the NFC library's binaries use RAM, meaning they do not allocate memory in the `data` and `bss` sections. A visualization is therefore omitted.

[1] https://github.com/N11cc00/master-thesis-benchmarks/commit/3bce32550b22466f493b146f72edc425ef9ac1a9

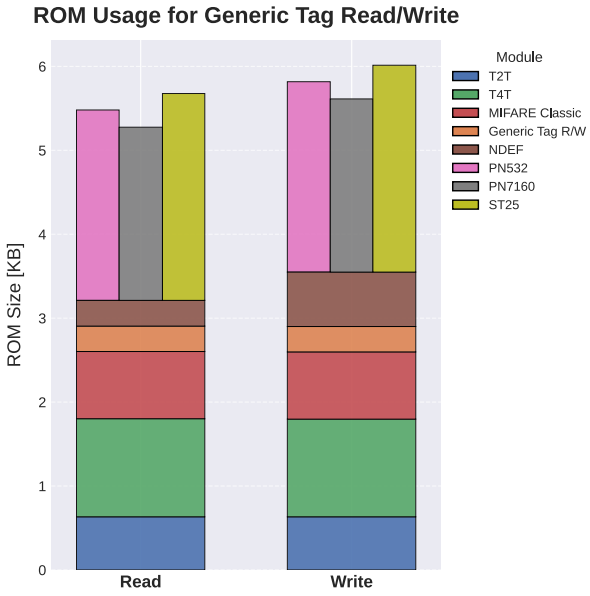

Figure 6.1: ROM usage for tag reading and writing of an NDEF message with the generic tag reader/writer.

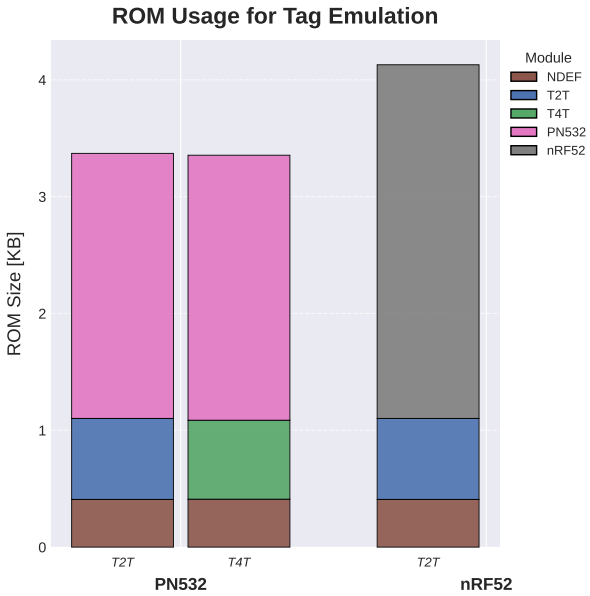

Figure 6.2: ROM usage for tag emulation. Both reading and writing are covered by the same test for tag emulation.

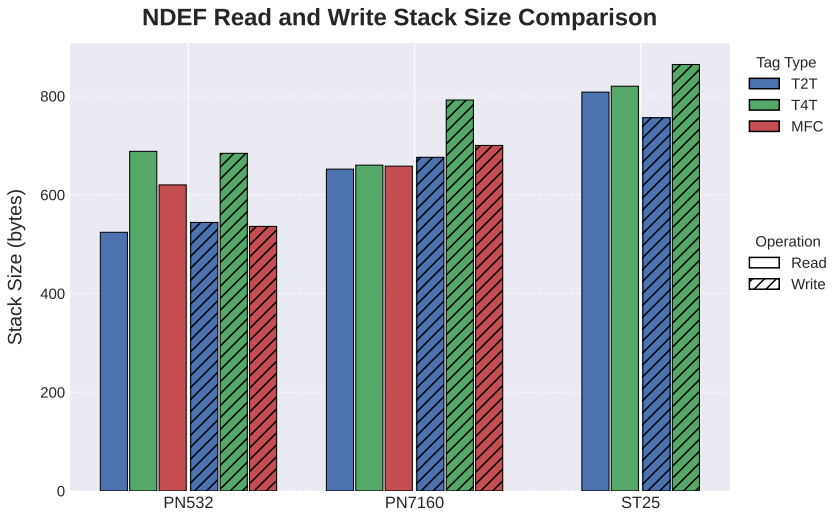

Figure 6.3: Maximum stack size used throughout the program for every test.

The ROM usage for the tag reading and writing is done by inspecting the binary of a test program using the generic reader and writer wrapper. The tests are slightly modified, but they mostly resemble the code displayed in Listing 1. For compilation, `gcc` is used.

The ROM usage is similar for both tag reading and writing. The generic tag reader/writer acts as a wrapper around all other tag libraries. This means that including the generic tag reader and writer also includes the other tag libraries, irrespective of actual usage. The tag writing requires a slightly bigger overhead as it adds the record to the NDEF message before writing it to the tag. The difference in ROM size for the tag libraries between read and write operations is minimal. The ST25R3916B has the biggest ROM size, as the anti-collision activity, as well as the ISO-DEP protocol, are part of the driver. The PN7160 driver implementation leads to a smaller footprint because the NCI handles more protocol functionality. The user can save on ROM size by using the specific tag library instead of relying on the generic wrapper, but this means that the user has to know the tag type at compile time.

For tag emulation, the nRF52840 only supports T2Ts. The T2T and T4T libraries have a similar size when used in tag emulation. The driver for the nRF52840 also includes the nrfx nfct driver by Nordic Semiconductor [30], which leads to a bigger ROM size for this NFC peripheral. The tag emulation is not split into reading and writing because a tag emulator is receptive to both operations at any time. This means that the ROM size is independent of the operation mode for tag emulation.

6.1.2 Stack Size Usage

The stack size usage can be seen in Figure 6.3 for tag reading and writing. It compares the maximum stack size usage for the entire duration of the program by using the `test_utils_print_stack_usage` RIOT module. The stack size includes other RIOT modules that are required to make the program compile.

The stack size for reading and writing T4Ts requires more of the stack than the other tag types for all device drivers. The reason might be the additional APDU parsing required for T4Ts. The stack size is bigger for the ST25, as more of the protocols are implemented in software. The

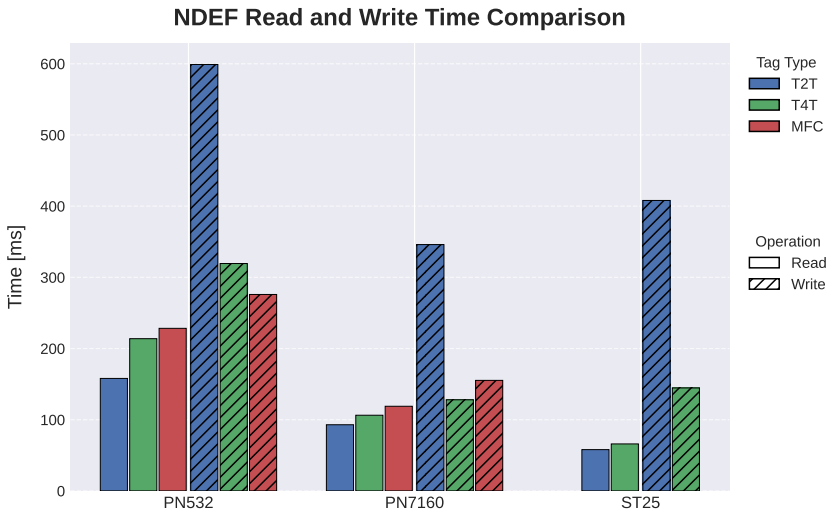

Figure 6.4: Timing for reading and writing of an NDEF message.

ST25 does not support MIFARE Classic tags. The stack sizes are comparable for tag reading and writing for all NFC devices.

6.2 Timing

The read and write times for an NDEF message are compared. The NDEF message has a total length of 200 bytes and contains a single text record. The timing tests are done with the `ztimer` RIOT module. The time is the duration it takes to execute the NDEF read and write function as part of the generic tag reader/writer. All tests were repeated 30 times, and the resulting standard deviation σ was smaller than $5\,\mathrm{ms}$. This equates to a relative deviation from the average of $5\,\%$. Hence, only the averages are displayed as they are accurate enough for the purpose of comparison.

6.2.1 Tag Reading and Writing

The results for each NFC peripheral and each tag type are compared in Figure 6.4. T2Ts have faster read access because they do not require ISO-DEP activation or authentication. While reading a T2T is comparatively fast, writing to them takes longer because of the small write size of 4 bytes instead of the 16-byte read commands. Unsurprisingly, writing takes four times longer than reading for T2Ts. The other tags are considerably faster when written to.

MFCs behaves similarly to read or write targets, but writing operations are split into two parts and are expected to take longer because three transmissions will have to be made for a single write. T4Ts are comparable to MFCs in both reading and writing modes. They cannot outperform MFCs despite the larger read and write instruction sizes allowing for more data to be sent in one NFC frame. A reason might be the extra ISO-DEP protocol operations and APDU processing.

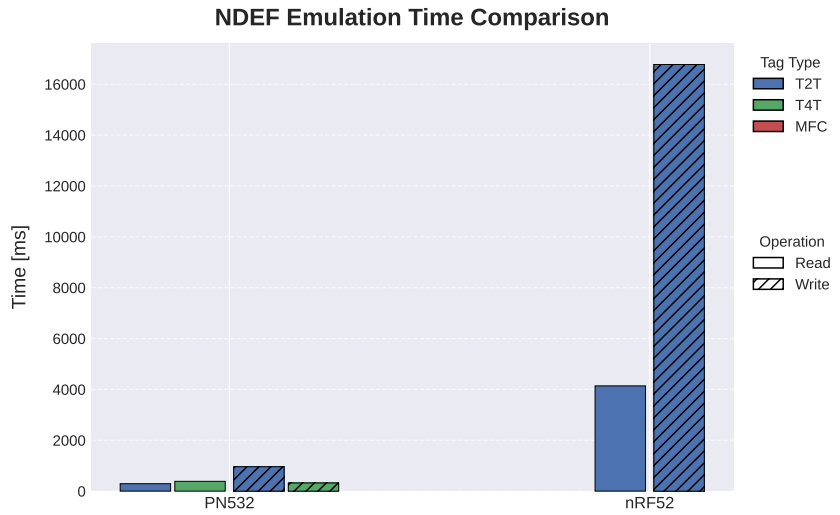

Figure 6.5: Timing for tag emulation.

6.2.2 Tag Emulation

The time for this experiment was taken by measuring the time it takes for an NFC reader and writer to either read or write the 200-byte NDEF message. The reader and writer used in this case is the PN532.

The comparison for tag emulation speeds can be seen in Figure 6.5. The nRF52840 takes substantially longer to be read from or written to than the PN532. This is because the driver emits IRQs for field lost events that should not occur, as outlined in chapter 5. As such, the current state of the driver implementation is unsuited for real-world use. The nRF52840 only supports T2Ts.

The PN532 has longer timings being written to than being read from when emulating a T2T. This matches the results in the previous read and write experiments. The T4T emulator has closer reading and writing speeds. The PN7160 and the ST25R3916B do not support the NFC listen modes with the current driver implementation.

6.2.3 Related NFC Libraries

The related NFC libraries compared in this chapter are the libraries that were more closely described in chapter 3. The experiments must be done on the same hardware, but nfcpy and libnfc do not run on the nRF52840. For this reason, the Raspberry Pi 1 B with an ARM32 architecture was selected. RIOT can be compiled on Linux and then runs natively as a user-space application. The common NFC device supported by all libraries is the PN532. Pull-up resistors are not supported in the native configuration on RIOT, and one $10\,\mathrm{k\Omega}$ resistor was added externally.

Another consideration is that nfcpy does not support the PN532's SPI interface and instead relies on UART. The RIOT driver for the PN532 only supports SPI. To make results comparable, the SPI speed was reduced to $100\,\mathrm{kHz}$ to match nfcpy's default UART baudrate of 115200.

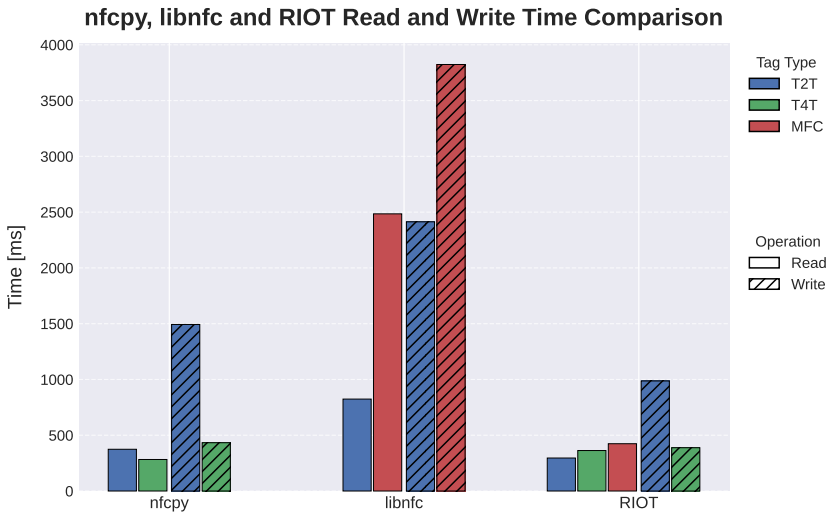

Figure 6.6: Related NFC library comparison for read and write.

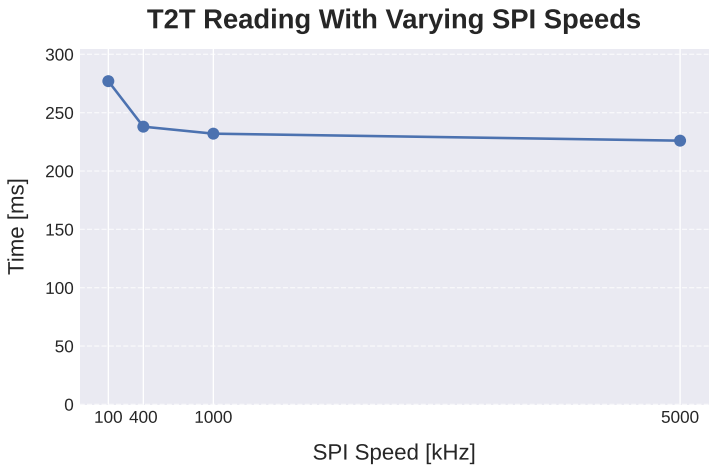

Figure 6.7: The impact of varying SPI speed on the overall timings. The PN532 was used to read a T2T in this test.

The comparison was done by reading or writing the entire NDEF message. The utilities provided by libnfc do not operate on NDEF messages but on the whole tag. Nfcpy behaves similarly to RIOT and only queries the necessary bytes on the tag. The comparison of the different libraries is found in Figure 6.6.

Libnfc produces the biggest times because it reads and writes the entire tag memory. Nfcpy is comparable to RIOT, with RIOT being slightly faster for T2Ts. Writing of MFCs does not work on the Raspberry Pi for the RIOT NFC library. Running the RIOT application as native on the Raspberry Pi 1 might cause timing issues for the MFC's stricter timings or misbehavior with the IRQ GPIO. The test does work on the nRF52840dk.

6.2.4 Serial Speed

To estimate the impact of the serial communication between the RIOT host device and the NFC device on the overall timing of the library, timings were taken with different SPI speeds and the PN532. The SPI speeds reflect all clock frequencies supported by RIOT and the PN532 used for testing. RIOT provides a higher $10\,\mathrm{MHz}$ clock, but the PN532 only supports SPI speeds up

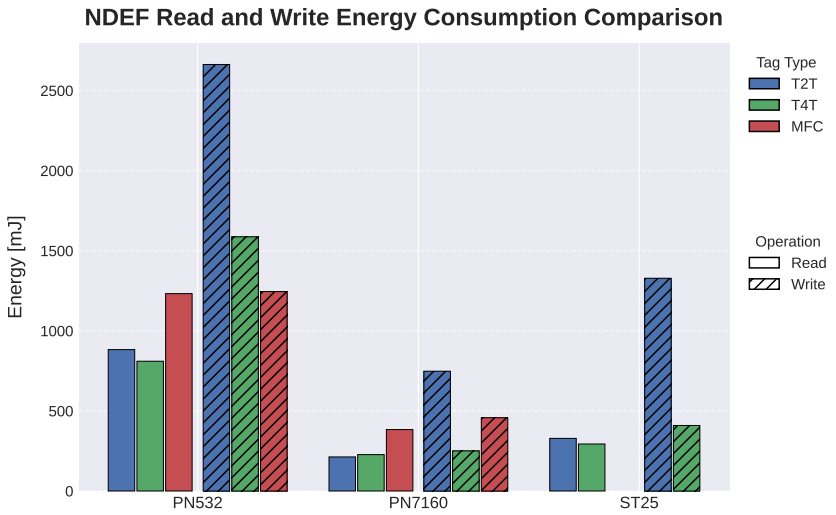

Figure 6.8: Energy usage for every read and write scenario.

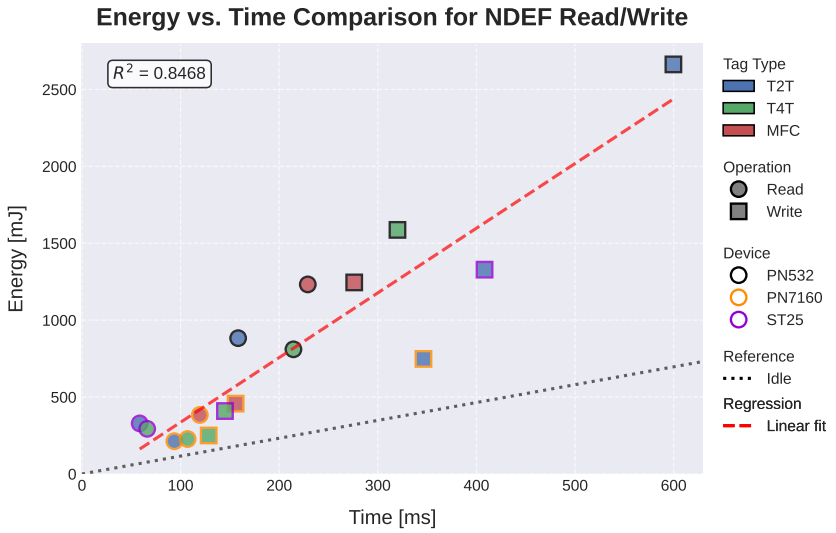

Figure 6.9: Time and energy comparison for every tag type, operation mode, and NFC device. One data point represents one test case. The gray line is the baseline energy consumption. The red line is the linear regression line. The top left shows the coefficient of determination R^2.

to $5\,\mathrm{MHz}$. The graph can be found in Figure 6.7. The time references a full read of a 200-byte NDEF message from a T2T.

Increasing the SPI speed has a diminishing effect on the overall speed of the application. The biggest jump can be seen between $100\,\mathrm{kHz}$ and $400\,\mathrm{kHz}$. At $1\,\mathrm{MHz}$, the 5-times increase of the clock frequency only speeds the application up by a few milliseconds. Under the assumption that higher speeds than $5\,\mathrm{MHz}$ have even less of an impact on the overall time, the communication overhead for $1\,\mathrm{MHz}$ can be estimated to be $\approx 10\,\mathrm{ms}$ by measuring the difference in time for $1\,\mathrm{MHz}$ and extrapolated higher SPI speeds.

6.3 Energy Usage

The section takes the experiments done in subsection 6.2.1 and measures the energy used by the nRF52840 and the NFC peripheral. For this, the Power Profiler Kit 2 by Nordic Semiconductor[2] was used. The experiments were repeated ten times with a supply voltage of $3\,\mathrm{V}$. The average energy consumption for one entire read or write operation can be seen in Figure 6.8. The standard deviation σ for all experiments is smaller than $18\,\mathrm{mJ}$. The maximum relative standard deviation is $4\,\%$.

The PN532 uses the most energy for all operations compared to the other devices. The PN7160 uses the least amount of energy for all operations and tag types.

The graph in figure Figure 6.9 compares the timing data taken from subsection 6.2.1 with the energy consumption for each device, tag type, and operation mode. There is a very strong positive correlation between time and energy, highlighted by the red line with a coefficient of determination R^2 of 0.85. The coefficient was calculated across all devices. Unsurprisingly, the longer it takes for an operation to complete, the more energy is used. It is still valuable to know that the correlation holds true regardless of the used NFC peripheral. The baseline power draw is marked by the gray line and sits at $1.6\,\mathrm{mW}$ without any of the NFC devices connected to the nRF52840 development kit. The baseline power draw includes the on-board LEDs and the J-Link debugger that must also be powered.

[2] https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2

7 Conclusion and Further Work

This thesis has demonstrated a successful design and implementation of a novel and comprehensive NFC library for the IoT. The library bridges multiple NFC devices (nRF52840, PN532, PN7160, and ST25R3916B) and achieves hardware abstraction without sacrificing device-specific features for various tag technologies (T2T, T4T, and MFC) as well as the two common modes.

A technical contribution is the `nfcdev` interface, which balances software flexibility with hardware capabilities. It was argued that the interface is better suited for an NFC library than a software-heavy approach or an implementation of the NCI for every device driver because it is simpler to implement new drivers and high-level libraries, and has a simpler interface. The interface allows operation on any NFC technology (A, B, F, V) with data exchange in both directions without prohibitive overhead.

The high-level libraries are compatible with protocol specifications and implement T2T, T4T, and MIFARE Classic support in reader and writer mode, as well as T2T and T4T emulation while making use of the new interface. On top of this, the libraries provide a way to directly manipulate the tag data structures. The NDEF library was expanded to accommodate NDEF message parsing. A design and implementation of P2P communication via the LLCP in the connectionless mode is also a part of the library.

The resource consumption and timings of the various use cases are within the bounds of real-world IoT applications. The library outperforms similar NFC libraries (libnfc, nfcpy) with the added benefit of being tailored for constrained devices. The feature set of the library also rivals other NFC libraries, with room for more protocol implementations. Tag support could be added for T3T or T5T.

Another area that can be considered for further research is encrypted NFC. This could pave the way for contactless payments according to the EMV specification [21].

Further implementations could include the SNEP and the TNEP for expanded P2P functionality. Other proprietary protocols for the MIFARE Ultralight or the MIFARE DESFire could add features like encryption or faster tag access. The existing device drivers of the PN7160 and the ST25R3916B can be examined further and completed to add working listen modes. Adding more device drivers can always be a consideration, of course.

Some limitations of the library include the lackluster nRF52840 NFC peripheral driver implementation, which is too slow for practical applications. The MIFARE Classic implementation only works with tags that use one key for encryption. The LLCP is only featured in the connectionless mode, missing the connection-oriented mode of the protocol. Listen mode is only supported by the drivers of the nRF52840 and the PN532.

The NFC library will be incrementally integrated into the RIOT codebase with multiple pull requests to the official RIOT GitHub repository. The pull requests for the old T2T emulation will be closed in favor of the new `nfcdev` interface[1]. The NDEF library implementation was only marginally modified for this thesis, and the existing pull request for the NDEF library will be tackled first[2].

[1] https://github.com/RIOT-OS/RIOT/pull/21240
[2] https://github.com/RIOT-OS/RIOT/pull/21272

Bibliography

[1] NFC Forum. *NFC Data Exchange Format (NDEF)*. July 24, 2006.

[2] NXP Semiconductors. *UM0701-02 PN532 User Manual*. 2nd edition. Nov. 5, 2007. URL: https://www.nxp.com/docs/en/user-guide/141520.pdf (visited on 11/14/2025).

[3] Bundesamt für Sicherheit in der Informationstechnik (BSI). *BSI TR-03105 Part 1.2 - Component Specification RFID*. Tech. rep. BSI - Bundesamt für Sicherheit in der Informationstechnik, Nov. 14, 2008. URL: https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Standards-und-Zertifizierung/Technische-Richtlinien/TR-nach-Thema-sortiert/tr03105/TR-03105_node.html (visited on 11/14/2025).

[4] NFC Forum. *NFC Forum Logical Link Control Protocol Specification*. Dec. 11, 2009.

[5] NFC Forum. *NFC Forum Simple NDEF Exchange Protocol Specification*. Aug. 31, 2011.

[6] NFC Forum. *NFC Forum Type 1 Tag Operation Specification*. Apr. 13, 2011.

[7] NFC Forum. *NFC Forum Type 2 Tag Operation Specification*. May 31, 2011.

[8] NFC Forum. *NFC Forum Type 3 Tag Operation Specification*. June 28, 2011.

[9] NFC Forum. *NFC Forum Type 4 Tag Operation Specification*. June 28, 2011.

[10] Japan Standards Association. *Contactless Integrated Circuit Cards — Part 4: High-speed Proximity Cards*. Tokyo, Japan: Japan Standards Association, 2016.

[11] Emmanuel Baccelli et al. "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT". In: *IEEE Internet of Things Journal* 5.6 (Mar. 2018). DOI: 10.1109/JIOT.2018.2815038. URL: https://www.riot-os.org/assets/pdfs/riot-ieeeiotjournal-2018.pdf.

[12] NFC Forum. *NFC Forum Type 5 Tag Operation Specification*. Apr. 27, 2018.

[13] ISO/IEC JTC 1/SC 17 Identifikationskarten et al. *ISO/IEC 14443-1 Cards and security devices for personal identification - Contactless proximity objects - Part 1: Physical characteristics*. Geneve, Apr. 2018.

[14] ISO/IEC JTC 1/SC 17 Identifikationskarten et al. *ISO/IEC 14443-3 Cards and security devices for personal identification - Contactless proximity objects - Part 3: Initialization and anticollision*. Geneve, July 2018.

[15] ISO/IEC JTC 1/SC 17 Identifikationskarten et al. *ISO/IEC 14443-4 Cards and security devices for personal identification - Contactless proximity objects - Part 4: Transmission protocol*. Geneve, July 2018.

[16] ISO/IEC JTC 1/SC 17 Identifikationskarten et al. *ISO/IEC 15693-1 Cards and security devices for personal identification - Contactless vicinity objects - Part 1: Physical characteristics*. Geneve, July 2018.

[17] ISO/IEC JTC 1 ISO/IEC-Gemeinschaftskomitee für Informationstechnik et al. *ISO/IEC 15693-2 Cards and security devices for personal identification - Contactless vicinity objects - Part 2: Air interface and initialization*. Geneve, Apr. 2019.

[18] ISO/IEC JTC 1/SC 17 Identifikationskarten et al. *ISO/IEC 14443-2 Cards and security devices for personal identification - Contactless proximity objects - Part 2: Radio frequency power and signal interface*. Geneve, July 2020.

[19] ISO/IEC JTC 1/SC 17 Identifikationskarten et al. *ISO/IEC 7816-4 Identification cards - Integrated circuit cards - Part 4: Organization, security and commands for interchange*. Geneve, May 2020.

[20] NFC Forum. *NFC Forum Digital Protocol Specification*. Aug. 3, 2021.

[21] EMVCo. *EMV Contactless Interface Specification*. Tech. rep. Version 3.2. EMVCo, July 2022. URL: https://www.emvco.com/specifications/ (visited on 12/01/2025).

[22] Stephen Tiedemann and contributors. *nfcpy*. Version 1.0.4. Mar. 10, 2022. URL: https://github.com/nfcpy/nfcpy (visited on 11/14/2025).

[23] ISO/IEC JTC 1/SC 6 Datenkommunikation et al. *ISO/IEC 18092 Telecommunications and information exchange between systems - Near Field Communication Interface and Protocol 1 (NFCIP-1)*. Geneve, Dec. 2023.

[24] NFC Forum. *NFC Forum Activity Specification*. Feb. 3, 2023.

[25] NFC Forum. *NFC Forum NFC Controller Interface*. June 22, 2023.

[26] STMicroelectronics. *X-CUBE-NFC6*. Version 2.2.0. July 15, 2024. URL: https://www.st.com/en/embedded-software/x-cube-nfc6.html (visited on 11/14/2025).

[27] ISO/IEC JTC 1/SC 17 Identifikationskarten et al. *ISO/IEC DIS 15693-3 Cards and security devices for personal identification - Contactless vicinity objects - Part 3: Anticollision and transmission protocol*. Geneve, May 2025.

[28] nfc-tools. *libnfc*. Version 3fa0751ad. Mar. 5, 2025. URL: https://github.com/nfc-tools/libnfc (visited on 11/15/2025).

[29] Nordic Semiconductor. *nRF Connect SDK*. Version 3.1.1. Sept. 17, 2025. URL: https://github.com/nrfconnect/sdk-nrf (visited on 11/15/2025).

[30] Nordic Semiconductor. *nrfx*. Version 4.0.0. Nov. 7, 2025. URL: https://github.com/NordicSemiconductor/nrfx (visited on 11/16/2025).

[31] NXP Semiconductors. *UM11495 PN7160 NFC controller User Manual*. 1.8 edition. June 2, 2025. URL: https://www.nxp.com/docs/en/user-manual/UM11495.pdf (visited on 11/14/2025).

[32] STMicroelectronics. *ST25R3916B Data Sheet. NFC reader for payment, consumer and industrial*. 10th edition. July 8, 2025. URL: https://www.st.com/resource/en/datasheet/st25r3916b.pdf (visited on 11/15/2025).

[33] Nordic Semiconductor. *nRF52840*. URL: https://www.nordicsemi.com/Products/nRF52840 (visited on 11/14/2025).

[34] RIOT OS Community. *RIOT OS - Supported Boards*. URL: https://www.riot-os.org/boards.html (visited on 11/15/2025).

[35] RIOT OS Community. *RIOT OS - Supported CPUs*. URL: https://www.riot-os.org/cpus.html (visited on 11/15/2025).

[36] RIOT OS Community. *RIOT OS - Supported Drivers*. URL: https://www.riot-os.org/drivers.html (visited on 11/15/2025).

[37] Sony. *FeliCa. Contactless IC Card Technology*. URL: https://www.sony.co.jp/en/Products/felica/ (visited on 11/21/2025).

List of Figures

List of Tables