nfcpy documentation

Release 1.0.4

Stephen Tiedemann

March 10, 2022

Contents

This documentation covers the '1.0.4' version of **nfcpy**. There are also other versions.

The **nfcpy** module implements NFC Forum specifications for wireless short-range data exchange with NFC devices and tags. It is written in Python and aims to provide an easy-to-use yet powerful framework for applications integrating NFC. The source code is licensed under the EUPL and hosted at GitHub. The latest release version can be installed from PyPI with `pip install -U nfcpy`.

To send a web link to a smartphone:

```
import nfc
import ndef
from threading import Thread

def beam(llc):
    snep_client = nfc.snep.SnepClient(llc)
    snep_client.put_records([ndef.UriRecord('http://nfcpy.org')])

def connected(llc):
    Thread(target=beam, args=(llc,)).start()
    return True

with nfc.ContactlessFrontend('usb') as clf:
    clf.connect(llcp={'on-connect': connected})
```

There are also a number of *Example Programs* that can be used from the command line:

```
$ examples/beam.py send link http://nfcpy.org
```

CHAPTER 1

Overview

1.1 Requirements

- Python version 2.7 or 3.5 or newer
- Python usb1 module to access USB devices through libusb
- Python serial module to access serial (incl. FTDI) devices
- Python docopt module for some of the example programs

1.2 Supported Devices

The contactless devices known to be working with *nfcpy* are listed below with the device path column showing the full *path* argument for the `nfc.clf.ContactlessFrontend.open()` method or the `--device` option that most example programs support. The testbed column shows the devices that are regularly tested with *nfcpy*.

Manufacturer	Product	NFC Chip	Device Path	Testbed	Notes
Sony	RC-S330	RC-S956	usb:054c:02e1	Yes	[1]
Sony	RC-S360	RC-S956	usb:054c:02e1	Yes	[1]
Sony	RC-S370	RC-S956	usb:054c:02e1	No	[1]
Sony	RC-S380/S	Port100	usb:054c:06c1	Yes	[2]
Sony	RC-S380/P	Port100	usb:054c:06c3	No	[2]
Sony	Board	PN531v4.2	usb:054c:0193	Yes	[3]
Philips/NXP	Board	PN531v4.2	usb:04cc:0531	Yes	[3]
Identive	SCL3710	PN531	usb:04cc:0531	No	[4]
ACS	ACR122U	PN532v1.4	usb:072f:2200	Yes	[5]
ACS	ACR122U	PN532v1.6	usb:072f:2200	Yes	[5]
Stollmann	Reader	PN532v1.4	tty:USB0:pn532	Yes	[6]
Adafruit	Board	PN532v1.6	tty:AMA0:pn532	Yes	[7]
Identive	SCL3711	PN533v2.7	usb:04e6:5591	Yes	[8]
Identive	SCL3712	PN533	usb:04e6:5593	No	[9]
SensorID	StickID	PN533v2.7	usb:04cc:2533	Yes	[10]
Arygon	ADRA	PN531v4.2	tty:USB0:arygon	Yes	

1.2.1 Functional Support

The following table summarizes the functional support level of the supported devices. Identical devices are aggregated under one of the product names. Only testbed devices are covered. In the table an `x` means that the function is supported by hardware and software while an `o` means that the hardware would support but but the software not yet implemented. More information about individual driver / hardware restrictions can be found in the *nfc.clf* documentation.

	Tag Read/Write					Tag Emulation					Peer2Peer		
	1	2	3	4A	4B	1	2	3	4A	4B	I	T	ac
RC-S380	x	x	x	x	x		o	x	o		x	x	
RC-S956		x	x	x	x		o		o		x	x	
PN533	x	x	x	x	x		o	x	o		x	x	x
PN532	x	x	x	x	x		o	x	o		x	x	x
PN531		x	x	x			o		o		x	x	x
ACR122U		x	x	x	x						x		

[1] The Sony RC-S330, RC-S360, and RC-S370 are in fact identical devicess, the difference is only in size and thus antenna.

[2] The only known difference between RC-S380/S and RC-S380/P is that the RC-380/S has the CE and FCC certification marks for sales in Europe and US.

[3] This is a reference board that was once designed by Philips and Sony and has a hardware switch to select either the Philips or Sony USB Vendor/Product ID. The chip can only handle Type A and Type F technology.

[4] This device is supported as a standard PN531. It has been reported to work as expected but is not part of regular testing.

[5] While the ACR122U internally uses a PN532 contactless chip the functionality provided by a PN532 can not be fully used due to an additional controller that implements a USB-CCID interface (for PC/SC) towards the host. It is possible using PCSC_Escape commands to unleash some functionality but this this is not equivalent to directly accessing a PN532. **It is not recommended to buy this device for use with nfcpy.**

[6] The path shown is for Ubuntu Linux in case the reader is the first UART/USB bridge found by the OS. Also on Windows OS the path is slightly different (`com:COM1:pn532`).

[7] This is sold by Adafruit as "PN532 NFC/RFID Controller Breakout Board" and can directly be connected to a serial port of, for example, a Raspberry Pi (the device path shown is for the Raspberry Pi's UART, when using a USB/UART bridge it would be `usb:USB0:pn532`). Note that the serial link speed is only 115200 baud when connected at /dev/ttyAMA0 while with a USB/UART bridge it may be up to 921600 baud (on Linux the driver tries to figure this out).

[8] The SCL3711 has a relatively small antenna that winds around the circuitry and may be the reason for less superior performance when operating as a target in passive communication mode (where the external field must be modulated).

[9] The SCL3712 has been reported to work but is not available for regular testing.

[10] The SensorID USB stick is a native PN533. It has no EEPROM attached and thus uses the default NXP Vendor/Product IDs from the ROM code. Absence of an EEPROM also means that the firmware uses default RF settings.

1.2.2 General Notes

- Testbed devices are verified to work with the latest stable nfcpy release. Test platforms are Ubuntu Linux (usually the latest version), Raspbian (with Raspberry Pi 2 Model B), and Windows (currently a Windows 7 virtual machine). No tests are done for MAC OS X because of lack of hardware.
- All device architectures with a PN532 or PN533 suffer from a firmware bug concerning Type 1 Tags with dynamic memory layout (e.g. the Topaz 512). With *nfcpy* version 0.10 this restriction could be removed by directly adressing the Contactless Interface Unit (CIU) within the chip.
- The ACR122U is not supported as P2P Target because the listen time can not be set to less than 5 seconds. It can not be overstated that the ACR122U is not a good choice for *nfcpy*.

1.3 Implementation Status

Specification	Status
TS NFC Digital Protocol 1.1	implemented
TS NFC Activity 1.1	implemented
TS Type 1 Tag Operation 1.2	implemented
TS Type 2 Tag Operation 1.2	implemented
TS Type 3 Tag Operation 1.2	implemented
TS Type 4 Tag Operation 3.0	implemented
TS NFC Data Exchange Format 1.0	except chunking
TS NFC Record Type Definition 1.0	implemented
TS Text Record Type 1.0	implemented
TS URI Record Type 1.0	implemented
TS Smart Poster Record Type 1.0	implemented
TS Signature Record Type	not implemented
TS Logical Link Control Protocol 1.3	implemented
TS Simple NDEF Exchange Protocol 1.0	implemented
TS Connection Handover 1.2	implemented
TS Personal Health Communication 1.0	implemented
AD Bluetooth Secure Simple Pairing	implemented

1.4 References

- NFC Forum Specifications: http://nfc-forum.org/our-work/specifications-and-application-documents/

CHAPTER 2

Getting started

2.1 Installation

NFCPy requires the library libusb for generic access to USB devices.

Install libusb (Linux)

Linux distributions usually have this installed, otherwise it should be available through the standard package manager (beware not to choose the old version `0.x`).

Install libusb (Windows)

Windows users will have to manually install WinUSB and libusb. Microsoft provides instructions to install WinUSB but a much simpler approach is to use Zadig (a driver installation helper application).

- Download Zadig
- Connect your NFC device
- Run the downloaded executable
- Click Options -> List All Devices
- Select your NFC reading/writing device from the list
- Select the WinUSB driver from the other drop down and install it

Then, install libusb:

- Download libusb (Downloads -> Latest Windows Binaries).
- Unpack the 7z archive (you may use 7zip).
- For 32-bit Windows:
  - Copy `MS32\dll\libusb-1.0.dll` to `C:\Windows\System32`.
- For 64-bit Windows:
  - Copy `MS64\dll\libusb-1.0.dll` to `C:\Windows\System32`.

- Copy `MS32\dll\libusb-1.0.dll` to `C:\Windows\SysWOW64`.

Install Python and nfcpy

Download and install Python (2.7 or 3.5 or later).

Note: Python may already be installed on your system if you are a Linux user.

Once Python is installed use pip to install the latest stable version of *nfcpy*. This will also install the required `libusb1` and `pyserial` Python modules.

```
$ pip install -U nfcpy
```

Windows users will want to ensure they have configured their environment's PATH correctly, otherwise they will not be able to access `pip` on the command line. It is usually located at `C:\Python27\Scripts\pip.exe` so they must ensure `C:\Python27\Scripts\` is on their PATH.)

Verify installation

Check if everything installed correctly and that *nfcpy* is able to find your contactless reader.

```
$ python -m nfc
```

If all goes well the output should tell that your your reader was found, below is an example of how it may look with an SCL3711:

```
This is the latest version of nfcpy run in Python 2.7.12
on Linux-4.4.0-47-generic-x86_64-with-Ubuntu-16.04-xenial
I'm now searching your system for contactless devices
** found SCM Micro SCL3711-NFC&RW PN533v2.7 at usb:002:024
I'm not trying serial devices because you haven't told me
-- add the option '--search-tty' to have me looking
-- but beware that this may break existing connections
```

Common problems on Linux (access rights or other drivers claiming the device) should be reported with a possible solution:

```
This is the latest version of nfcpy run in Python 2.7.12
on Linux-4.4.0-47-generic-x86_64-with-Ubuntu-16.04-xenial
I'm now searching your system for contactless devices
** found usb:04e6:5591 at usb:002:025 but access is denied
-- the device is owned by 'root' but you are 'stephen'
-- also members of the 'root' group would be permitted
-- you could use 'sudo' but this is not recommended
-- it's better to add the device to the 'plugdev' group
   sudo sh -c 'echo SUBSYSTEM==\"usb\", ACTION==\"add\", ATTRS{idVendor}==\"04e6\",␣
↪ATTRS{idProduct}==\"5591\", GROUP=\"plugdev\" >> /etc/udev/rules.d/nfcdev.rules'
   sudo udevadm control -R # then re-attach device
I'm not trying serial devices because you haven't told me
-- add the option '--search-tty' to have me looking
-- but beware that this may break other serial devs
Sorry, but I couldn't find any contactless device
```

2.2 Open a local device

Any data exchange with a remote NFC device needs a contactless frontend attached and opened for communication. Most commercial devices (also called NFC Reader) are physically attached through USB and either provide a native USB interface or a virtual serial port.

The *nfc.ContactlessFrontend* manages all communication with a local device. The *open* method tries to find and open a device and returns True for success. The string argument determines the device with a sequence of components separated by colon. The first component determines where the device is attached (usb, tty, or udp) and what the further components may be. This is best explained by example.

Suppose a FeliCa S330 Reader is attached to a Linux computer on USB bus number 3 and got device number 9 (note that device numbers always increment when a device is connected):

```
$ lsusb
...
Bus 003 Device 009: ID 054c:02e1 Sony Corp. FeliCa S330 [PaSoRi]
...
```

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend()
>>> assert clf.open('usb:003:009') is True     # open device 9 on bus 3
>>> assert clf.open('usb:054c:02e1') is True   # open first PaSoRi 330
>>> assert clf.open('usb:003') is True         # open first Reader on bus 3
>>> assert clf.open('usb:054c') is True        # open first Sony Reader
>>> assert clf.open('usb') is True             # open first USB Reader
>>> clf.close()  # previous open calls implicitly closed the device
```

Some devices, especially for embedded projects, have a UART interface that may be connected either directly or through a USB UART adapter. Below is an example of a Raspberry Pi 3 which has two UART ports (ttyAMA0, ttyS0) and one reader is connected with a USB UART adapter (ttyUSB0). On a Raspberry Pi 3 the UART linked from /dev/serial1 is available on the GPIO header (the other one is used for Bluetooth connectivity). On a Raspberry Pi 2 it is always ttyAMA0.

```
pi@raspberrypi ~ $ ls -l /dev/tty[ASU]* /dev/serial?
lrwxrwxrwx 1 root root          5 Dez 21 18:11 /dev/serial0 -> ttyS0
lrwxrwxrwx 1 root root          7 Dez 21 18:11 /dev/serial1 -> ttyAMA0
crw-rw---- 1 root dialout 204, 64 Dez 21 18:11 /dev/ttyAMA0
crw-rw---- 1 root dialout   4, 64 Dez 21 18:11 /dev/ttyS0
crw-rw---- 1 root dialout 188,  0 Feb 24 12:17 /dev/ttyUSB0
```

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend()
>>> assert clf.open('tty:USB0:arygon') is True  # open /dev/ttyUSB0 with arygon driver
>>> assert clf.open('tty:USB0:pn532') is True   # open /dev/ttyUSB0 with pn532 driver
>>> assert clf.open('tty:AMA0') is True         # try different drivers on /dev/
↪ttyAMA0
>>> assert clf.open('tty') is True              # try all serial ports and drivers
>>> clf.close()  # previous open calls implicitly closed the device
```

A special kind of device bus that does not require any physical hardware is provided for testing and application prototyping. It works by sending NFC communication frames across a UDP/IP connection and can be used to connect two processes running an *nfcpy* application either locally or remote.

In the following example the device path is supplied as an init argument. This would raise an `exceptions.IOError` with `errno.ENODEV` if it fails to open. The example also demonstrates the use of a `with` statement for automatic close when leaving the context.

```
>>> import nfc
>>> with nfc.ContactlessFrontend('udp') as clf:
...     print(clf)
...
Linux IP-Stack on udp:localhost:54321
```

2.3 Read and write tags

NFC Tag Devices are tiny electronics devices with a comparatively large (some square centimeters) antenna that serves as both an inductive power receiver and for communication. The energy is provided by the NFC Reader Device for as long as it wishes to communicate with the Tag.

Most Tags are embedded in plastics or paper and can store data in persistent memory. NFC Tags as defined by the NFC Forum have standardized memory format and command set to store NFC Data Exchange Format (NDEF) records. Most commercial NFC Tags also provide vendor-specific commands for special applications, some of those can be used with *nfcpy*. A rather new class of NFC Interface Tags is targeted towards providing NFC communication for embedded devices where the information exchange is through NFC with the microcontroller of the embedded device.

Tip: It is quite easy to make an NFC field detector. Just a few turns of copper wire around three fingers and the ends soldered to an LED will do the job. Here's a video.

NFC Tags are simple slave devices that wait unconditionally for any reader command to respond. This makes it easy to interact with them from within a Python interpreter session using the local contactless frontend.

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend('usb')
```

The *clf.sense()* method can now be used to search for a proximity target with arguments set for the desired communication technologies. The example shows the result of a Type F card response for which the `nfc.tag.activate()` function then returns a *Type3Tag* instance.

```
>>> from nfc.clf import RemoteTarget
>>> target = clf.sense(RemoteTarget('106A'), RemoteTarget('106B'), RemoteTarget('212F
↪'))
>>> print(target)
212F sensf_res=0101010701260CCA020F0D23042F7783FF12FC
>>> tag = nfc.tag.activate(clf, target)
>>> print(tag)
Type3Tag 'FeliCa Standard (RC-S960)' ID=01010701260CCA02 PMM=0F0D23042F7783FF SYS=12FC
```

The same *Type3Tag* instance can also be acquired with the *clf.connect()* method. This is the generally preferred way to discover and activate contactless targets of any supported type. When configured with the *rdwr* dictionary argument the *clf.connect()* method will use Reader/Writer mode to discover NFC Tags. When a Tag is found and activated, the `on-connect` callback function returning `False` means that the tag presence loop shall not be run but the *nfc.tag.Tag* object returned immediately. A more useful callback function could do something with the *tag* and return `True` for requesting a presence loop that makes *clf.connect()* return only after the tag is gone.

```
>>> tag = clf.connect(rdwr={'on-connect': lambda tag: False})
>>> print(tag)
Type3Tag 'FeliCa Standard (RC-S960)' ID=01010701260CCA02 PMM=0F0D23042F7783FF SYS=12FC
```

An NFC Forum Tag can store NFC Data Exchange Format (NDEF) Records in a specifically formatted memory region. NDEF data is found automatically and wrapped into an *NDEF* object accessible through the *tag.ndef*

attribute. When NDEF data is not present the attribute is simply `None`.

```
>>> assert tag.ndef is not None
>>> for record in tag.ndef.records:
...     print(record)
...
NDEF Uri Record ID '' Resource 'http://nfcpy.org'
```

The *tag.ndef.records* attribute contains a list of NDEF Records decoded from *tag.ndef.octets* with the ndeflib package. Each record has common and type-specific methods and attributes for content access.

```
>>> record = tag.ndef.records[0]
>>> print(record.type)
urn:nfc:wkt:U
>>> print(record.uri)
http://nfcpy.org
```

A list of NDEF Records assigned to *tag.ndef.records* gets encoded and then written to the Tag (internally the bytes are assigned to *tag.ndef.octets* to trigger the update).

```
>>> import ndef
>>> uri, title = 'http://nfcpy.org', 'nfcpy project'
>>> tag.ndef.records = [ndef.SmartposterRecord(uri, title)]
```

When NDEF data bytes are written to a Memory Tag then the *tag.ndef* object matches the stored data. In case of an Interface Tag this may not be true because the write commands may be handled differently by the device. The only way to find out is read back the data and compare. This is the logic behind *tag.ndef.has_changed*, which should be `False` for a Memory Tag.

```
>>> assert tag.ndef.has_changed is False
```

An NFC Interface Tag may be used to realize a device that presents dynamically changing NDEF data depending on internal state, for example a sensor device returning the current temperature.

```
>>> tag = clf.connect(rdwr={'on-connect': lambda tag: False})
>>> print(tag)
Type3Tag 'FeliCa Link (RC-S730) Plug Mode' ID=03FEFFFFFFFFFFFF PMM=00E1000000FFFF00␣
↪SYS=12FC
>>> assert tag.ndef is not None and tag.ndef.length > 0
>>> assert tag.ndef.records[0].type == 'urn:nfc:wkt:T'
>>> print('Temperature 0: {}'.format(tag.ndef.records[0].text))
Temperature 0: +21.3 C
>>> for count in range(1, 4):
...     while not tag.ndef.has_changed: time.sleep(1)
...     print('Temperature {}: {}'.format(count, tag.ndef.records[0].text))
...
Temperature 1: +21.0 C
Temperature 2: +20.5 C
Temperature 3: +20.1 C
```

Finally the contactless frontend should be closed.

```
>>> clf.close()
```

Documentation of all available Tag classes as well as NDEF class methods and attributes can be found in the *nfc.tag* module reference. For NDEF Record class types, methods and attributes consult the ndeflib documentation.

2.4 Emulate a card

It is possible to emulate a card (NFC Tag) with *nfcpy* but unfortunately this only works with some NFC devices and is limited to Type 3 Tag emulation. The RC-S380 fully supports Type 3 Tag emulation. Devices based on PN532, PN533, or RC-S956 chipset can also be used but an internal frame size limit of 64 byte only allows read/write operations with up to 3 data blocks.

Below is an example of an NDEF formatted Type 3 Tag. The first 16 byte (first data block) contain the attribute data by which the reader will learn the NDEF version, the number of data blocks that can be read or written in a single command, the total capacity and the write permission state. Bytes 11 to 13 contain the current NDEF message length, initialized to zero. The example is made to specifically open only an RC-S380 contactless frontend (otherwise the number of blocks that may be read or written should not be more than 3).

```
import nfc
import struct

ndef_data_area = bytearray(64 * 16)
ndef_data_area[0] = 0x10  # NDEF mapping version '1.0'
ndef_data_area[1] = 12    # Number of blocks that may be read at once
ndef_data_area[2] = 8     # Number of blocks that may be written at once
ndef_data_area[4] = 63    # Number of blocks available for NDEF data
ndef_data_area[10] = 1    # NDEF read and write operations are allowed
ndef_data_area[14:16] = struct.pack('>H', sum(ndef_data_area[0:14]))  # Checksum

def ndef_read(block_number, rb, re):
    if block_number < len(ndef_data_area) / 16:
        first, last = block_number*16, (block_number+1)*16
        block_data = ndef_data_area[first:last]
        return block_data

def ndef_write(block_number, block_data, wb, we):
    global ndef_data_area
    if block_number < len(ndef_data_area) / 16:
        first, last = block_number*16, (block_number+1)*16
        ndef_data_area[first:last] = block_data
        return True

def on_startup(target):
    idm, pmm, sys = '03FEFFE011223344', '01E0000000FFFF00', '12FC'
    target.sensf_res = bytearray.fromhex('01' + idm + pmm + sys)
    target.brty = "212F"
    return target

def on_connect(tag):
    print("tag activated")
    tag.add_service(0x0009, ndef_read, ndef_write)
    tag.add_service(0x000B, ndef_read, lambda: False)
    return True

with nfc.ContactlessFrontend('usb:054c:06c1') as clf:
    while clf.connect(card={'on-startup': on_startup, 'on-connect': on_connect}):
        print("tag released")
```

This is a fully functional NFC Forum Type 3 Tag. With a separate reader or Android apps such as NXP Tag Info and NXP Tag Writer, NDEF data can now be written into the **ndef_data_area** and read back until the loop is terminated with `Control-C`.

2.5 Work with a peer

The best part of NFC comes when the limitations of a single master controlling a humble servant are overcome. This is achieved by the NFC Forum Logical Link Control Protocol (LLCP), which allows multiplexed communications between two NFC Forum Devices with either peer able to send protocol data units at any time and no restriction to a single application run in one direction.

An LLCP link between two NFC devices is requested with the **llcp** argument to *clf.connect()*.

```
>>> import nfc
>>> clf = ContactlessFrontend('usb')
>>> clf.connect(llcp={}) # now touch a phone
True
```

When the first example got LLCP running there is actually just symmetry packets exchanged back and forth until the link is broken. We have to use callback functions to add some useful stuff.

```
>>> def on_connect(llc):
...     print llc; return True
...
>>> clf.connect(llcp={'on-connect': connected})
LLC: Local(MIU=128, LTO=100ms) Remote(MIU=1024, LTO=500ms)
True
```

The on_connect function receives a single argument **llc**, which is the *LogicalLinkController* instance coordinates aal data exchange with the remote peer. With this we can add client applications but they must be run in a separate execution context to have on_connect return fast. Only after on_connect returns, the **llc** can start running the symmetry loop (the LLCP heartbeat) with the remote peer and generally receive and dispatch protocol and service data units.

When using the interactive interpreter it is less convinient to program in the callback functions so we will start a thread in the callback to execute the *llc.run** loop and return with False. This tells *clf.connect()* to return immediately with the **llc** instance).

```
>>> import threading
>>> def on_connect(llc):
...     threading.Thread(target=llc.run).start(); return False
...
>>> llc = clf.connect(llcp={'on-connect': on_connect})
>>> print llc
LLC: Local(MIU=128, LTO=100ms) Remote(MIU=1024, LTO=500ms)
```

Application code is not supposed to work directly with the **llc** object but use it to create *Socket* objects for the actual communication. Two types of regular sockets can be created with either `nfc.llcp.LOGICAL_DATA_LINK` for a connection-less socket or `nfc.llcp.DATA_LINK_CONNECTION` for a connection-mode socket. A connection-less socket does not guarantee that application data is delivered to the remote application (although *nfcpy* makes sure that it's been delivered to the remote device). A connection-mode socket cares about reliability, unless the other implementation is buggy data you send is guaranteed to make it to the receiving application - error-free and in order.

What can be done with an Android phone as the peer device is for example to send to its default SNEP Server. SNEP is the NFC Forum Simple NDEF Exchange Protocol and a default SNEP Server is built into Android under the name of Android Beam. SNEP messages are exchanged over an LLCP data link connection so we create a connection mode socket, connect to the server with the service name known from the NFC Forum Assigned Numbers Register and then send a SNEP PUT request with a web link to open.

```
>>> import ndef
>>> socket = nfc.llcp.Socket(llc, nfc.llcp.DATA_LINK_CONNECTION)
>>> socket.connect('urn:nfc:sn:snep')
>>> records = [ndef.UriRecord("http://nfcpy.org")]
>>> message = b''.join(ndef.message_encoder(records))
>>> socket.send(b"\x10\x02\x00\x00\x00" + chr(len(message)) + message)
>>> socket.recv()
'\x10\x81\x00\x00\x00\x00'
>>> socket.close()
```

The phone should now have opened the http://nfcpy.org web page.

The code can be simplified by using the *SnepClient* from the *nfc.snep* package.

```
>>> import nfc.snep
>>> snep = nfc.snep.SnepClient(llc)
>>> snep.put_records([ndef.UriRecord("http://nfcpy.org")])
True
```

The `put()` method is smart enough to temporarily connect to `urn:nfc.sn:snep` for sending. There are also methods to open and close the connection explicitely and maybe use a different service name.

Note: The *Logical Link Control Protocol* tutorial has more information on LLCP in general and how its used with *nfcpy*. The *nfc.llcp* package documentation contains describes all the API classes and methods that are available.

CHAPTER 3

Logical Link Control Protocol

The Logical Link Control Protocol allows multiplexed communications between two NFC Forum Peer Devices where either peer can send protocol data units at any time (asynchronous balanced mode). The communication endpoints are called Service Access Points (SAP) and are addressed by a 6 bit numerical identifier. Protocol data units are exchanged between exactly two service access points, from a source SAP (SSAP) to a destination SAP (DSAP). The service access point address space is split into 3 parts: an address between 0 and 15 identifies a well-known service, an address between 16 and 31 identifies a service that is registered in the local service environment, and addresses between 32 and 63 are unregistered and normally used as a source address by client applications that send or connect to peer services.

The interface to realize LLCP client and server applications in nfcpy is implemented by the *nfc.llcp.Socket* class. A socket is created with a *LogicalLinkController* instance and the *socket type* as arguments to the *Socket* constructor. The `nfc.ContactlessFrontend.connect()` method accepts callback functions that will receive the active *LogicalLinkController* instance as argument.

```
import nfc
import nfc.llcp

def client(socket):
    socket.sendto("message", addr=16)

def connected(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.LOGICAL_DATA_LINK)
    Thread(target=client, args=(socket,)).start()
    return True

clf = nfc.ContactlessFrontend()
clf.connect(llcp={'on-connect': connected})
```

Although service access points are generally identified by a numerical address, the LLCP service discovery component allows SAPs to be associated with a globally unique service name and become discoverable by remote applications. A service name may represent either an NFC Forum well-known or an externally defined service name.

- The format `urn:nfc:sn:<servicename>` represents a well-known service name, for example the service name `urn:nfc:sn:snep` identifies the NFC Forum Simple NDEF Data Exchange (SNEP) default server.

- The format `urn:nfc:xsn:<domain>:<servicename>` represents a service name that is defined by the *domain* owner, for example the service name `urn:nfc:xsn:nfc-forum.org:snep-validation` is the service name of a special SNEP server used by the NFC Forum during validation of the SNEP secification.

In nfcpy a service name can be registered with *Socket.bind()* and a service name string as the address parameter. The allocated service access point address number can then be retrived with *getsockname()*. A remote service name can be resolved into a service access point address number with *resolve()*.

```
def server(socket):
    message, address = socket.recvfrom()
    socket.sendto("It's me!", address)

def client(socket):
    address = socket.resolve( 'urn:nfc:xsn:nfcpy.org:test-service' )
    socket.sendto("Hi there!", address)
    message, address = socket.recvfrom()
    print("SAP {0} said: {1}".format(address, message))

def startup(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.LOGICAL_DATA_LINK)
    socket.bind( 'urn:nfc:xsn:nfcpy.org:test-service' )
    print("server bound to SAP {0}".format(socket.getsockname()))
    Thread(target=server, args=(socket,)).start()
    return llc

def connected(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.LOGICAL_DATA_LINK)
    Thread(target=client, args=(socket,)).start()
    return True

clf = nfc.ContactlessFrontend()
clf.connect(llcp={'on-startup': startup, 'on-connect': connected})
```

Connection-mode sockets must be connected before data can be exchanged. For a server socket this involves calls to *bind()*, *listen()* and *accept()*, and for a client socket to call *resolve()* and *connect()* with the address returned by *resolve()* or to simply call *connect()* with the service name as *address* (note that *resolve()* becomes more efficient when queries for multiple service names are needed).

```
def server(socket):
    # note that this server only accepts one connection
    # for multiple connections spawn a thread per accept
    while True:
       client = socket.accept()
       while True:
           message = client.recv()
           print("Client said: {0}".format(message))
           client.send("It's me!")

def client(socket):
    socket.connect( 'urn:nfc:xsn:nfcpy.org:test-service' )
    socket.send("Hi there!")
    message = socket.recv()
    print("Server said: {0}".format(message))

def startup(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.DATA_LINK_CONNECTION)
    socket.bind( 'urn:nfc:xsn:nfcpy.org:test-service' )
    print("server bound to SAP {0}".format(socket.getsockname()))
```

(continues on next page)

(continued from previous page)

```
    socket.listen()
    Thread(target=server, args=(socket,)).start()
    return llc

def connected(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.DATA_LINK_CONNECTION)
    Thread(target=client, args=(socket,)).start()
    return True

clf = nfc.ContactlessFrontend()
clf.connect(llcp={'on-startup': startup, 'on-connect': connected})
```

Data can be send and received with *sendto()* and *recvfrom()* on connection-less sockets and *send()* and *recv()* on connection-mode sockets. Send data is guaranteed to be delivered to the remote device when the send methods return (although not necessarily to the remote service access point - only for a connection-mode socket this can be safely assumed but note that even then data may not yet have been arrived at the service user). Receiving data with either *recv()* or *recvfrom()* blocks until some data has become available or all LLCP communication has been terminated (if either one peer intentionally closes the LLCP Link or the devices are moved out of communication range). To implement a communication timeout during normal operation, the *poll()* method can be used to waI will "fix" this bug by adding to the documentationI will "fix" this bug by adding to the documentationit for a 'recv' event with a given timeout.

```
def client(socket):
    socket.connect( 'urn:nfc:xsn:nfcpy.org:test-service' )
    socket.send("Hi there!")
    if socket.poll('recv', timeout=1.0):
        message = socket.recv()
        print("Server said: {0}".format(message))
    else:
        print("Server said nothing within 1 second")
```

Sockets of type `nfc.llcp.LOGICAL_DATA_LINK`, `DATA_LINK_CONNECTION` and `RAW_ACCESS_POINT` (which should normally not be used) do not provide fragmentation for messages that do not fit into a single protocol data unit but raise an `nfc.llcp.Error` exception with `errno.EMSGSIZE`. An application can learn the maximum nuber of bytes for sending or receiving by calling *getsockopt()* with option `nfc.llcp.SO_SNDMIU` or `nfc.llcp.SO_RCVMIU`.

```
send_miu = socket.getsockopt(nfc.llcp.SO_SNDMIU)
recv_miu = socket.getsockopt(nfc.llcp.SO_RCVMIU)
```

When opening or accepting a data link connection an application may specify the maximum number of octets to receive with the `nfc.llcp.SO_RCVMIU` option in *setsockopt()*. The value must be between 128 and 2176, inclusively. If the RCVMIU is not explicitely set for a data link connection the default value applied by the peer is 128 octets.

On connection-mode sockets options `nfc.llcp.SO_SNDBUF` and `nfc.llcp.SO_RCVBUF` can be used to learn the local and remote receive window values established during connection setup. The local receive window can also be set with *setsockopt()* before the socket gets connected.

```
def server(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.DATA_LINK_CONNECTION)
    socket.setsockopt(nfc.llcp.SO_RCVMIU, 1000)
    socket.setsockopt(nfc.llcp.SO_RCVBUF, 2)
    socket.bind( "urn:nfc:sn:snep" )
    socket.listen()
```

(continues on next page)

(continued from previous page)

```
    socket.accept()
    ...

def client(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.DATA_LINK_CONNECTION)
    socket.setsockopt(nfc.llcp.SO_RCVMIU, 1000)
    socket.setsockopt(nfc.llcp.SO_RCVBUF, 2)
    socket.connect( "urn:nfc:sn:snep" )
    ...
```

LLCP data link connections use sliding window flow-control. The receive window set with `nfc.llcp.SO_RCVBUF` dictates the number of connection-oriented information PDUs that the remote side of the data link connection may have outstanding (sent but not acknowledged) at any time. A connection-mode socket is able to receive and buffer that number of packets. Whenever the service user (the application) retrieves one or more messages from the socket, reception of the messages will be acknowledged to the remote SAP.

A common application architecture is that messages are received in a dedicated thread and then added to a message queue that the application will query for data to process at a later time. Unless the message queue can grow indefinitely it may happen that the receive thread is unable to add more data to the queue because the application is not consuming data for some reason. For such situations LLCP provides a mechanism to convey a *busy* indication to the remote service user. In nfcpy an application uses *`setsockopt()`* with option `nfc.llcp.SO_RCVBSY` and value `True` to set the *busy* state or value `False` to clear the *busy* state. An application can use *`getsockopt()`* with option `nfc.llcp.SO_RCVBSSY` to learn it's own *busy* state and `nfc.llcp.SO_SNDBSY` to learn the remote application's *busy* state.

CHAPTER 4

Simple NDEF Exchange Protocol

The NFC Forum Simple NDEF Exchange Protocol (SNEP) allows two NFC devices to exchange NDEF Messages. It is implemented in many smartphones and typically used to push phonebook contacts or web page URLs to another phone.

SNEP is a stateless request/response protocol. The client sends a request to the server, the server processes that request and returns a response. On the protocol level both the request and response have no consequences for further request/response exchanges. Information units transmitted through SNEP are NDEF messages. The client may use a SNEP PUT request to send an NDEF message and a SNEP GET request to retrieve an NDEF message. The message to retrieve with a GET request depends on an NDEF message sent with the GET request but the rules to determine equivalence are an application layer contract and not specified by SNEP.

NDEF messages can easily be larger than the maximum information unit (MIU) supported by the LLCP data link connection that a SNEP client establishes with a SNEP server. The SNEP layer handles fragmentation and reassembly so that an application must not be concerned. To avoid exhaustion of the limited NFC bandwidth if an NDEF message would exceed the SNEP receiver's capabilities, the receiver must acknowledge the first fragment of an NDEF message that can not be transmitted in a single MIU. The acknowledge can be either the request/response codes CONTINUE or REJECT. If CONTINUE is received, the SNEP sender shall transmit all further fragments without further acknowledgement (the LLCP data link connection guarantees successful transmission). If REJECT isreceived, the SNEP sender shall abort tranmsission. Fragmentation and reassembly are handled transparently by the *nfc.snep.SnepClient* and *nfc.snep.SnepServer* implementation and only a REJECT would be visible to the user.

A SNEP server may return other response codes depending on the result of a request:

- A SUCCESS response indicates that the request has succeeded. For a GET request the response will include an NDEF message. For a PUT request the response is empty.
- A NOT FOUND response says that the server has not found anything matching the request. This may be a temporary of permanent situation, i.e. the same request send later could yield a different response.
- An EXCESS DATA response may be received if the server has found a matching response but sending it would exhaust the SNEP client's receive capabilities.
- A BAD REQUEST response indicates that the server detected a syntax error in the client's request. This should almost never be seen.

- The NOT IMPLEMENTED response will be returned if the client sent a request that the server has not implemented. It applies to existing as well as yet undefined (future) request codes. The client can learn the difference from the version field transmitted withnthe response, but in reality it doesn't matter - it's just not supported.
- With UNSUPPORTED VERSION the server reacts to a SNEP version number sent with the request that it doesn't support or refuses to support. This should be seen only if the client sends with a higher major version number than the server has implemented. It could be received also if the client sends with a lower major version number but SNEP servers are likely to support historic major versions if that ever happens (the current SNEP version is 1.0).

Besides the protocol layer the SNEP specification also defines a *Default SNEP Server* with the well-known LLCP service access point address 4 and service name `urn:nfc:sn:snep`. Certified NFC Forum Devices must have the *Default SNEP Server* implemented. Due to that requirement the feature set and guarantees of the *Default SNEP Server* are quite limited - it only implements the PUT request and the NDEF message to put could be rejected if it is more than 1024 octets, though smartphones generally seem to support more.

4.1 Default Server

A basic *Default SNEP Server* can be built with *nfcpy* like in the following example (where all error and exception handling has been sacrified for brevity).

```
import nfc
import nfc.snep

class DefaultSnepServer(nfc.snep.SnepServer):
    def __init__(self, llc):
        nfc.snep.SnepServer.__init__(self, llc, "urn:nfc:sn:snep")

    def process_put_request(self, ndef_message):
        print("client has put an NDEF message")
        for record in ndef_message:
            print(record)
        return nfc.snep.Success

def startup(llc):
    global my_snep_server
    my_snep_server = DefaultSnepServer(llc)
    return llc

def connected(llc):
    my_snep_server.start()
    return True

my_snep_server = None
clf = nfc.ContactlessFrontend("usb")
clf.connect(llcp={'on-startup': startup, 'on-connect': connected})
```

This server will accept PUT requests with NDEF messages up to 1024 octets and return NOT IMPLEMENTED for any GET request. To increase the size of NDEF messages that can be received, the *max_ndef_message_recv_size* parameter can be passed to the *nfc.snep.SnepServer* class.

```
class DefaultSnepServer(nfc.snep.SnepServer):
    def __init__(self, llc):
        nfc.snep.SnepServer.__init__(self, llc, "urn:nfc:sn:snep", 10*1024)
```

4.2 Using SNEP Put

The *nfc.snep.SnepClient* provides two methods to send an NDEF message to the *Default SNEP Server*. A list of ndef.Record objects can be send with *nfc.snep.SnepClient.put_records()*. This encodes the records into a sequence of octets that are then send with *nfc.snep.SnepClient.put_octets()*.

The example below shows how the function to send the NDEF message is started as a separate thread - it cannot be directly called in connected() because the main thread context is used to operate the LLCP link.

```
import nfc
import nfc.snep
import threading

def send_ndef_message(llc):
    sp = ndef.SmartposterRecord('http://nfcpy.org', 'nfcpy home')
    nfc.snep.SnepClient(llc).put_records( [sp] )

def connected(llc):
    threading.Thread(target=send_ndef_message, args=(llc,)).start()
    return True

clf = nfc.ContactlessFrontend("usb")
clf.connect(llcp={'on-connect': connected})
```

Some phones require that a SNEP be present even if they are not going to send anything (Windows Phone 8 is such example). The solution is to also run a SNEP server on urn:nfc:sn:snep which will accept but discard SNEP Put requests from the peer device.

```
import nfc
import nfc.snep
import threading

server = None

def send_ndef_message(llc):
    sp_record = ndef.SmartposterRecord('http://nfcpy.org', 'nfcpy home')
    nfc.snep.SnepClient(llc).put_records( [sp_record] )

def startup(clf, llc):
    global server
    server = nfc.snep.SnepServer(llc, "urn:nfc:sn:snep")
    return llc

def connected(llc):
    server.start()
    threading.Thread(target=send_ndef_message, args=(llc,)).start()
    return True

clf = nfc.ContactlessFrontend("usb")
clf.connect(llcp={'on-startup': startup, 'on-connect': connected})
```

4.3 Private Servers

The SNEP protocol can be used for other, non-standard, communication between a server and client component. A private server can be run on a dynamically assigned service access point if a private service name is used. A private

server may also implement the GET request if it defines what a GET shall mean other than to return something. Below is an example of a private SNEP server that implements bot PUT and GET with the simple contract that whatever is put to the server will be returned for a GET request that requests the same or empty NDEF type and name values (for anything else the answer is NOT FOUND).

```
import nfc
import nfc.snep

class PrivateSnepServer(nfc.snep.SnepServer):
    def __init__(self, llc):
        self.ndef_message = [ndef.Record()]
        service_name = "urn:nfc:xsn:nfcpy.org:x-snep"
        nfc.snep.SnepServer.__init__(self, llc, service_name, 2048)

    def process_put_request(self, ndef_message):
        print("client has put an NDEF message")
        self.ndef_message = ndef_message
        return nfc.snep.Success

    def process_get_request(self, ndef_message):
        print("client requests an NDEF message")
        if ndef_message[0].type and ndef_message[0].type != self.ndef_message[0].type:
            return nfc.snep.NotFound
        if ndef_message[0].name and ndef_message[0].name != self.ndef_message[0].name:
            return nfc.snep.NotFound
        return self.ndef_message

def startup(clf, llc):
    global my_snep_server
    my_snep_server = PrivateSnepServer(llc)
    return llc

def connected(llc):
    my_snep_server.start()
    return True

my_snep_server = None
clf = nfc.ContactlessFrontend("usb")
clf.connect(llcp={'on-startup': startup, 'on-connect': connected})
```

A client application knowing the private server above may then use PUT and GET to set an NDEF message on the server and retrieve it back. The example code below also shows how results other than SUCCESS must be catched in try-except clauses. Note that *max_ndef_msg_recv_size* parameter is a policy sent to the SNEP server with every GET request.

```
import nfc
import nfc.snep
import threading

def send_ndef_message(llc):
    sp_record = ndef.SmartposterRecord('http://nfcpy.org', 'nfcpy home')
    snep = nfc.snep.SnepClient(llc, max_ndef_msg_recv_size=2048)
    snep.connect("urn:nfc:xsn:nfcpy.org:x-snep")
    snep.put( [sp_record] )

    print("*** get whatever the server has ***")
    print(snep.get_records( [ndef.Record()] ))
```

(continues on next page)

(continued from previous page)

```
    print("*** get a smart poster record ***")
    print(snep.get( [ndef.Record("urn:nfc:wkt:Sp")] ))

    print("*** get something that isn't there ***")
    try:
        snep.get( [ndef.Record("urn:nfc:wkt:Uri")] )
    except nfc.snep.SnepError as error:
        print(repr(error))

def connected(llc):
    threading.Thread(target=send_ndef_message, args=(llc,)).start()
    return True

clf = nfc.ContactlessFrontend("usb")
clf.connect(llcp={'on-connect': connected})
```

CHAPTER 5

Example Programs

tagtool.py Read or write or format tags for NDEF use.

beam.py Exchange NDEF data with a smartphone.

sense.py Sense for contactless targets.

listen.py Listen as contactless target.

rfstate.py Observe the RF field presence.

5.1 tagtool.py

The **tagtool.py** example program can be used to read or write NFC Forum Tags. For some tags, currently Type 3 Tags only, **tagtool** can also be used to format for NDEF use.

```
$ tagtool.py [-h|--help] [options] command
```

5.1.1 Options

--loop, -l
Repeat the command endlessly, use Control-C to abort.

--wait
After reading or writing a tag, wait until it is removed before returning. This option is implicit when the option `--loop` is set.

--technology `{A,B,F}`
Poll only for tags of a specific technology. The technologies NFC-A, NFC-B, and NFC-F are defined in the NFC Forum Digital Specification. The technology indicator is case insensitive. The default is to poll for all technologies.

-q
Do not print log messages except for errors and warnings.

-d `MODULE`
Output debug messages for MODULE to the log facility. Logs are written to <stderr> unless a log file is set with `-f`. MODULE is a string that corresponds to an *nfcpy* module or individual file, with dots between path components. For example, `-d nfc` enables all *nfcpy* debug logs, `-d nfc.tag` enables debug logs for all tag types, and `-d nfc.tag.tt3` enables debug logs only for type 3 tags. This option may be given multiple times to enable debug logs for several modules.

-f `LOGFILE`
Write debug log messages to <LOGFILE> instead of <stderr>. Info, warning and error logs will still be printed to <stderr> unless `-q` is set to supress info messages on <stderr>.

--nolog-symm
When operating in peer mode this option prevents logging of LLCP Symmetry PDUs from the `nfc.llcp.llc` module. Symmetry PDUs are exchanged regularly and quite frequently over an LLCP Link and are logged by default if debug output is enabled for the llcp module.

--device `PATH`
Use a specific reader or search only for a subset of readers. The syntax for PATH is:

- `usb[:vendor[:product]]` with optional *vendor* and *product* as four digit hexadecimal numbers, like `usb:054c:06c3` would open the first Sony RC-S380 reader and `usb:054c` the first Sony reader.
- `usb[:bus[:device]]` with optional *bus* and *device* number as three-digit decimal numbers, like `usb:001:023` would specifically mean the usb device with bus number 1 and device id 23 whereas `usb:001` would mean to use the first available reader on bus number 1.
- `tty:port:driver` with mandatory *port* and *driver* name should be used on Posix systems to open the serial port at device node `/dev/tty<port>` and load the driver from module `nfc/dev/<driver>.py`. A typical example would be `tty:USB0:arygon` for the Arygon APPx/ADRx at `/dev/ttyUSB0`.
- `com:port:driver` with mandatory *port* and *driver* name should be used on Windows systems to open the serial port `COM<port>` and load the `nfc/dev/<driver>.py` driver module.
- `udp[:host][:port]` with optional *host* name or address and *port* number will use a fake communication channel over UDP/IP. Either value may be omitted in which case *host* defaults to 'localhost' and *port* defaults to 54321.

-p `PASSWORD`
Use PASSWORD to authentication with a tag that supports password protection. This would be the same password as used in *`tagtool.py protect -p`* to set a password.

5.1.2 Commands

Available commands are listed below. The default if no command is specified is to invoke **tagtool.py show**.

show

The **show** command prints information about a tag, including NDEF data if present.:

```
$ tagtool.py [options] show [-h] [-v]
```

-v
Print verbose information about the tag found. The amount of additional information depends on the tag type.

dump

The **dump** command dumps tag data to the console or into a file. Data written to the console is a hexadecimal string. Data written to a file is raw bytes.

```
$ tagtool.py [options] dump [-h] [-o FILE]
```

-o FILE
Write data to FILE. Data format is plain bytes.

load

The **load** command writes data to a tag. Data may be plain bytes or a hex string, as generated by the **dump** command or with the **ndeftool**.

```
$ tagtool.py [options] load [-h] FILE
```

FILE
Load NDEF data to write from **FILE** which must exist and be readable. The file may contain NDEF data in either raw bytes or a hexadecimal string which gets converted to bytes. If **FILE** is specified as a single dash - data is read from **stdin**.

format

The **format** command writes NDEF capability information for an empty NDEF memory area on NFC Forum compliant tags. A tag type may be specified to give further options.

```
$ tagtool.py [options] format [-h] [options] {tt1,tt2,tt3,tt4} ...
```

--version x.y
The format of the management information that describes the NDEF data area on the tag, as defined in the NFC Forum tag specifications. Only defined version numbers are acceptable. The version must be expressed as a version string of the form <major>.<minor>, where each component is an integer between 0 and 15, inclusively. For example, --version 1.3 denotes major version 1 and minor version 3. If --version is not provided, the highest possible version number is used.

--wipe BYTE
When formatting a tag the NDEF message data itself is usually not touched and could be easily recovered. The --wipe options instructs the formatter to overwrite the complete data area with the given 8-bit integer value. Depending on the tag type and size this may take a couple of seconds.

format tt1

The **format tt1** command formats the NDEF partition on a Type 1 Tag.

```
$ tagtool.py [options] format tt1 [-h]
```

--magic BYTE
: The value to use as the NDEF magic byte. This option can be used to set an invalid magic byte.

--ver x.y
: Type 1 Tag NDEF mapping version number, specified as a version string in the same way as for to the `--version` argument. The difference is that this version number will be written regardless of whether it constitutes a valid version number.

--tms BYTE
: Value to write into the tag memory size byte.

--rwa BYTE
: Value to write into the read/write access byte.

format tt2

The **format tt2** command formats the NDEF partition on a Type 2 Tag.

```
$ tagtool.py [options] format tt2 [-h]
```

format tt3

The **format tt3** command formats the NDEF partition on a Type 3 Tag. With no additional options it does format for the maximum capacity. With further options it is possible to create any kind of weird tag formats for testing reader implementations. Note that none of these options is verified, except for the possible value range to fit the destination field. None of the options is necessary to create a correct format.

```
$ tagtool.py [options] format tt3 [-h] [--ver STR] [--nbr INT] [--nbw INT]
                                  [--max INT] [--rfu INT] [--wf INT]
                                  [--rw INT] [--len INT] [--crc INT]
```

--ver x.y
: Type 3 Tag NDEF mapping version number, specified as a version string in the same way as for to the `--version` argument. The difference is that this version number will be written regardless of whether it constitutes a valid version number.

--nbr N
: Type 3 Tag attribute block *Nbr* field value, the number of blocks that can be read at once. Must be an 8-bit integer in decimal or hexadecimal notation.

--nbw N
: Type 3 Tag attribute block *Nbw* field value, the number of blocks that can be written at once. Must be an 8-bit integer in decimal or hexadecimal notation.

--max N
: Type 3 Tag attribute block *Nmaxb* field value, which is the maximum number of blocks available for NDEF data. Must be a 16-bit integer in decimal or hexadecimal notation.

--rfu N
: Type 3 Tag attribute block *reserved* field value. Must be an 8-bit integer in decimal or hexadecimal notation.

--wf N
: Type 3 Tag attribute block *WriteF* field value. Must be an 8-bit integer in decimal or hexadecimal notation.

--rw N
: Type 3 Tag attribute block *RW Flag* field value. Must be an 8-bit integer in decimal or hexadecimal notation.

--len N
: Type 3 Tag attribute block *Ln* field value that specifies the actual size of the NDEF data stored. Must be a 24-bit integer in decimal or hexadecimal notation.

--crc N
: Type 3 Tag attribute block *Checksum* field value. Must be a 16-bit integer in decimal or hexadecimal notation. If not specified, the checksum is computed to be correct.

format tt4

The **format tt4** command formats the NDEF partition on a Type 4 Tag.

```
$ tagtool.py [options] format tt4 [-h]
```

protect

The **protect** command attempts to protect the tag against write modifications, optionally also against unauthorized read access. Support for protection depends on the tag type and product. Without options the the default attempt is protect with lock bits, be warned that this can not be undone. Lock bits are only available for type 1 and type 2 tags. With option `-p` the protection will be based on a password and further modifications are possible for anyone in posession of the password. Password protection works on NXP NTAG 21x type 2 tags and Sony FeliCa Lite-S type 3 tags.

```
$ tagtool.py protect [-h] [-p PASSWORD] [--from BLOCK] [--unreadable]
```

-p PASSWORD
: Protect the tag with the given PASSWORD. This works only for the NXP NTAG 21x type 2 tags and Sony FeliCa Lite-S type 3 tags. The password string is used as a key to compute an HMAC-SHA256 with the tag identifier (UID or IDm) as the message. The final password is the leftmost number of octets as needed for the tag product, 6 octets for an NTAG 21x and 16 octets for a FeliCa Lite-S. A password protected tag can then be unlocked with *tagtool.py -p*.

```
$ tagtool.py protect -p "my secret password"
$ tagtool.py -p "my secret password" protect -p "new secret"
```

--from BLOCK
: Start protecting data from a given block number. This option does only make sense on tags that organize memory in blocks or pages (Type 1, 2 and 3 Tags). A block corresponds to 4 byte of memory (a page) on Type 1 and 2 Tags, and 16 byte of memory on Type 3 Tags. If the tag has fewer blocks than specified, the value is silently adjusted to the largest possible.

--unreadable
: This option can only be used with password based protection. The result is that the tag will become unreadable without a password, i.e. the content is completely hidden. Further reads must then use the password option before the command.

```
$ tagtool.py -p "secret password" show
```

emulate

The **emulate** command emulates an NDEF tag if the hardware and driver support that functionality. The tag type must be specified following the optional parameters. The only currently supported tag type it **tt3**.

```
$ tagtool.py emulate [-h] [-l] [-k] [-s SIZE] [-p FILE] [FILE] {tt3} ...
```

FILE
Initialize the tag with NDEF data read from **FILE**. If not specified the tag will be just empty.

-l, --loop
Automatically restart after the tag has been released by the Initiator.

-k, --keep
If the `--loop` option is set, keep the same memory content after tag relase for the next tag activation. Without the `-k` option the tag memory is initialized from the command options for every activation.

-s SIZE
The minimum size for NDEF data. Depending on the tag type this may be rounded up to the nearest multiple of the tag storage granularity. If NDEF data is provided the size may be adjusted to fit the length of the data.

-p FILE
Preserve memory content in **FILE** after the tag is relased by the Initiator. The file is created if it does not exist and otherwise overwritten.

emulate tt3

The **emulate tt3** command emulates an NFC Forum Type 3 Tag.

```
$ tagtool.py [options] emulate [options] tt3 [-h] [--idm HEX] [--pmm HEX]
                                             [--sys HEX] [--bitrate {212,424}]
```

--idm HEX
The Manufacture Identifier to use in the polling response. Specified as a hexadecimal string. Defaults to `03FEFFE011223344`.

--pmm HEX
The Manufacture Parameter to use in the polling response. Specified as a hexadecimal string. Defaults to `01E0000000FFFF00`.

--sys HEX, **--sc** HEX
The system code use in the polling response if requested. Specified as a hexadecimal string. Defaults to `12FC`.

--bitrate {212,424}
The bitrate to listen for and respond with. Must be either 212 or 424. Defaults to 212 kbps.

5.1.3 Examples

Copy NDEF from one tag to another:

```
$ tagtool.py dump -o /tmp/tag.ndef && tagtool load /tmp/tag.ndef
```

Copy NDEF from one tag to many others:

```
$ tagtool.py dump -o /tmp/tag.ndef && tagtool --loop load /tmp/tag.ndef
```

5.2 beam.py

The **beam.py** example program uses the Simple NDEF Exchange Protocol (SNEP) to send or receive NDEF messages to or from a peer device, in most cases this will be a smartphone. The name *beam* is inspired by *Android Beam* and thus **beam.py** will be able to receive most content sent through *Android Beam*. It will not work for data that *Android Beam* sends with connection handover to Bluetooth or Wi-Fi, this may become a feature in a later version. Despite it's name, **beam.py** works not only with Android phones but any NFC enabled phone that implements the NFC Forum Default SNEP Server, such as Blackberry and Windows Phone 8.

```
$ beam.py [-h|--help] [OPTIONS] {send|recv} [-h] [OPTIONS]
```


5.2.1 Options

--loop, -l
Repeat the command endlessly, use Control-C to abort.

--mode `{t,i}`
Restrict the choice of NFC-DEP connection setup role to either `Target` (only listen) or `Initiator` (only poll). If this option is not given the dafault is to alternate between both roles with a randomized listen time.

--miu `INT`
Set a specific value for the LLCP Link MIU. The dafault value is 2175 octets.

--lto `INT`
Set a specific LLCP Link Timeout value. The default link timeout is 500 milliseconds.

--listen-time `INT`
Set the time to listen for initialization command from an NFC-DEP Initiator. The default listen time is 250 milliseconds.

--no-aggregation
Disable outbound packet aggregation for LLCP, i.e. do not generate LLCP AGF PDUs if multiple packets are waiting to be send. This is mostly to achieve communication with some older/buggy implementations.

-q
Do not print log messages except for errors and warnings.

-d MODULE
Output debug messages for MODULE to the log facility. Logs are written to <stderr> unless a log file is set with `-f`. MODULE is a string that corresponds to an *nfcpy* module or individual file, with dots between path components. For example, `-d nfc` enables all *nfcpy* debug logs, `-d nfc.tag` enables debug logs for all tag types, and `-d nfc.tag.tt3` enables debug logs only for type 3 tags. This option may be given multiple times to enable debug logs for several modules.

-f LOGFILE
Write debug log messages to <LOGFILE> instead of <stderr>. Info, warning and error logs will still be printed to <stderr> unless `-q` is set to supress info messages on <stderr>.

--nolog-symm
When operating in peer mode this option prevents logging of LLCP Symmetry PDUs from the `nfc.llcp.llc` module. Symmetry PDUs are exchanged regularly and quite frequently over an LLCP Link and are logged by default if debug output is enabled for the llcp module.

--device PATH
Use a specific reader or search only for a subset of readers. The syntax for PATH is:

- `usb[:vendor[:product]]` with optional *vendor* and *product* as four digit hexadecimal numbers, like `usb:054c:06c3` would open the first Sony RC-S380 reader and `usb:054c` the first Sony reader.
- `usb[:bus[:device]]` with optional *bus* and *device* number as three-digit decimal numbers, like `usb:001:023` would specifically mean the usb device with bus number 1 and device id 23 whereas `usb:001` would mean to use the first available reader on bus number 1.
- `tty:port:driver` with mandatory *port* and *driver* name should be used on Posix systems to open the serial port at device node `/dev/tty<port>` and load the driver from module `nfc/dev/<driver>.py`. A typical example would be `tty:USB0:arygon` for the Arygon APPx/ADRx at `/dev/ttyUSB0`.
- `com:port:driver` with mandatory *port* and *driver* name should be used on Windows systems to open the serial port `COM<port>` and load the `nfc/dev/<driver>.py` driver module.
- `udp[:host][:port]` with optional *host* name or address and *port* number will use a fake communication channel over UDP/IP. Either value may be omitted in which case *host* defaults to 'localhost' and *port* defaults to 54321.

5.2.2 Commands

send

Send an NDEF message to the peer device. The message depends on the positional argument that follows the *send* command and additional data.

```
$ beam.py send [--timeit] {link,text,file,ndef} [-h] [OPTIONS]
```

--timeit
Measure and print the time that was needed to send the message.

send link

Send a hyperlink embedded into a smartposter record.

```
$ beam.py send link URI [TITLE]
```

URI
The resource identifier, for example `http://nfcpy.org`.

TITLE
The smartposter title, for example `"nfcpy project home"`.

send text

Send plain text embedded into an NDEF Text Record. The default language identifier `en` can be changed with the `--lang` flag.

```
$ beam.py send text TEXT [OPTIONS]
```

TEXT
The text string to send.

--lang STRING
The language code to use when constructing the NDEF Text Record.

send file

Send a data file. This will construct a single NDEF record with *type* and *name* set to the file's mime type and path name, and the payload containing the file content. Both record type and name can also be explicitly set with the options `-t` and `-n`, respectively.

```
$ beam.py send file FILE [OPTIONS]
```

FILE
The file to send.

-t STRING
Set the record type. See the ndeflib for how to specify record types in *nfcpy*.

-n STRING
Set the record name (identifier).

send ndef

Send an NDEF message read from file. The file may contain multiple messages and if it does, then the strategy to select a specific message for sending can be specified with the `--select STRATEGY` option. For strategies that select a different message per touch beam.py must be called with the `--loop` flag. The strategies `first`, `last` and `random` select the first, or last, or a random message from FILE. The strategies `next` and `cycle` start with the first message and then count up, the difference is that `next` stops at the last message while `cycle` continues with the first.

```
$ beam.py send ndef FILE [OPTIONS]
```

FILE
The file from which to read NDEF messages.

--select STRATEGY
The strategy for NDEF message selection, it may be one of `first`, `last`, `next`, `cycle`, `random`.

recv

Receive an NDEF message from the peer device. The next positional argument determines what is done with the received message.

```
$ beam.py [OPTIONS] recv {print,save,echo,send} [-h] [OPTIONS]
```

recv print

Print the received message to the standard output stream.

```
$ beam.py recv print
```

recv save

Save the received message into a file. If the file already exists the message is appended.

```
$ beam.py recv save FILE
```

FILE
Name of the file to save messages received from the remote peer. If the file exists any new messages are appended.

recv echo

Receive a message and send it back to the peer device.

```
$ beam.py recv echo
```

recv send

Receive a message and send back a corresponding message if such is found in the *translations* file. The *translations* file must contain an even number of NDEF messages which are sequentially read into inbound/outbound pairs to form a translation table. If the receved message corresponds to any of the translation table inbound messages the corresponding outbound message is then sent back.

```
$ beam.py [OPTIONS] recv send [-h] TRANSLATIONS
```

TRANSLATIONS
A file with a sequence of NDEF messages.

5.2.3 Examples

Get a smartphone to open the nfcpy project page (which in fact just points to the code repository and documentation).

```
$ beam.py send link http://nfcpy.org "nfcpy project home"
```

Send the source file `beam.py`. On an Android phone this should pop up the "new tag collected" screen and show that a `text/x-python` media type has been received.

```
$ beam.py send file beam.py
```

The file `beam.py` is about 11 KB and may take some time to transfer, depending on the phone hardware and software. With a Google Nexus 10 it takes as little as 500 milliseconds while a Nexus 4 won't do it under 2.5 seconds.

```
$ beam.py send --timeit file beam.py
```

Receive a single NDEF message from the peer device and save it to *message.ndef* (note that if *message.ndef* exists the received data will be appended):

```
$ beam.py recv save message.ndef
```

With the `--loop` option it gets easy to collect messages into a single file.

```
$ beam.py --loop recv save collected.ndef
```

A file that contains a sequence of request/response message pairs can be used to send a specific response message whenever the associated request message was received.

```
$ echo -n "this is a request message" > request.txt
$ ndeftool.py pack -n '' request.txt -o request.ndef
$ echo -n "this is my reponse message" > response.txt
$ ndeftool.py pack -n '' response.txt -o response.ndef
$ cat request.ndef response.ndef > translation.ndef
$ beam.py recv send translation.ndef
```

5.3 sense.py

The **sense** example demonstrates the use of the *nfc.clf.ContactlessFrontend.sense()* method to discover contactless targets.

```
$ sense.py [target [target ...]] [options]
```

The *target* arguments define the type, bitrate and optional attributes for the contactless targets that may be discovered in a single sense loop. An empty loop (no targets) is allowed but is only useful to verify the *nfc.clf.ContactlessFrontend.sense()* method behavior. Optional arguments allow to set an iteration count and interval, continously repeat the (iterated) loop after a wait time, activate standard or verbose debug logs, and to specify the local device to use.

A *target* is specified by bitrate and a type identifier `A`, `B`, `F`. The following example would first sense for a DEP Target at 106kbps (in active communication mode), then for a Type A Target at 106 kbps, a Type B Target at 106kbps and a Type F Target at 212kbps.

```
$ sense.py 106A 106B 212F
```

Additional parameters can be supplied as comma-delimited name=value pairs in brackets. The example below searches for a 106 kbps DEP Target (in active communication mode) and then changes communication speed to 424 kbps.

```
$ sense.py '106A(atr_req=d400FFFFFFFFFFFFFFFF62260000003246666d010110)'

$ sense.py 106A --atr d400FFFFFFFFFFFFFFFF62260000003246666d010110
```

5.3.1 Options

-h, --help
Show a help message and exit.

--dep params
Attempt a DEP Target activation in passive communication mode when an appropriate Type A or Type F Target was discovered in in the main sense loop. The *params* argument defines optional attributes for the `nfc.clf.DEP` target object. The example below would try a DEP Target activation (in passive communication mode) with a parameter change to 424 kbps after 106 kbps Type A Target discovery.

```
$ sense.py 106A --dep 'psl_req=D404001203'
```

-i number
Specifies the number of iterations to run (default is 1 iteration). Each iteration is a sense for all the targets given as positional arguments.

-t seconds
The time between two iterations (default is 0.2 sec). It is measured from the start of one iteration to the start of the next iteration, effectively it will thus never be shorter than the execution time of an iteration.

-r, --repeat
Forever repeat the sense loop (including the number of iterations). Execution can be terminated with Ctrl-C.

-w seconds
Wait the specified number of seconds between repetitions (the default wait time is 0.1 sec).

-d, --debug
Activate debug log messages on standard error output.

-v, --verbose
Activate more debug log messages, most notably all commands send to the local device will be logged as well as their responses.

--device path
Specify a local device search path (the default is `usb`). For device path construction rules see *nfc.clf.ContactlessFrontend.open()*.

5.4 listen.py

Listen as Target for activation requests from a remote Initiator.

Usage:

```
listen.py tt2 [options] [--uid UID]
listen.py tt3 [options] [--idm <idm>] [--pmm <pmm>] [--sys <sys>]
listen.py tt4 [options] [--uid <uid>]
listen.py dep [options] [--id3 <id3>] [--gbt <gbt>] [--hce]
listen.py -h | --help
```

As the Target selected with the first positional argument listen.py waits '–time T' seconds for activation by a remote device and prints the local target configuration if not timed out. The listen period is repeated after '–wait T' seconds if the '–repeat' flag is given.

Without the '–repeat' flag, the exit status is 0 when activated and 1 if timed out, with the '–repeat' flag it is 0 for termination by keyboard interrupt (Ctrl-C). For argument errors and unsupported targets listen.py exits with 2. If a local device is not found or was removed listen.py exits with 3.

Options:

-h, --help	show this help message and exit
-t, --time T	listen time in seconds [default: 2.5]
-w, --wait T	time between repetations [default: 1.0]
-r, --repeat	repeat forever (cancel with Ctrl-C)
-d, --debug	output debug log messages to stderr
-v, --verbose	print and log more information
--device PATH	local device search path [default: usb]
--bitrate BR	set bitrate (default is 106 for A/B and 212 for F)
--uid UID	tt2/tt4 identifier [default: 08010203]
--idm IDM	tt3 identifier [default: 02FE010203040506]
--pmm PMM	tt3 parameters [default: FFFFFFFFFFFFFFFF]
--sys SYS	tt3 system code [default: 12FC]
--id3 ID3	dep nfcid3 [default: 01FE0102030405060708]
--gbt GBT	dep general bytes [default: 46666D010111]
--hce	announce dep and tt4 support for Type A

Examples:

```
listen.py tt2 --uid 08ABCDEF  # listen as Type 2 Tag with this UID
listen.py tt3 --bitrate 424   # listen as Type 3 Tag at 424 kbps
listen.py tt3 --sys 0003      # use the Suica system code for FeliCa
listen.py dep --gbt ''        # send ATR response without general bytes
listen.py dep --hce           # offer NFC-DEP Protocol and Type 4A Tag
```

Source:

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# -----------------------------------------------------------------------------
# Copyright 2015 Stephen Tiedemann <stephen.tiedemann@gmail.com>
#
# Licensed under the EUPL, Version 1.1 or - as soon they
# will be approved by the European Commission - subsequent
# versions of the EUPL (the "Licence");
# You may not use this work except in compliance with the
# Licence.
# You may obtain a copy of the Licence at:
#
# https://joinup.ec.europa.eu/software/page/eupl
#
# Unless required by applicable law or agreed to in
# writing, software distributed under the Licence is
# distributed on an "AS IS" basis,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied.
# See the Licence for the specific language governing
# permissions and limitations under the Licence.
# -----------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
"""Listen as Target for activation requests from a remote Initiator.

**Usage:** ::

  listen.py tt2 [options] [--uid UID]
  listen.py tt3 [options] [--idm <idm>] [--pmm <pmm>] [--sys <sys>]
  listen.py tt4 [options] [--uid <uid>]
  listen.py dep [options] [--id3 <id3>] [--gbt <gbt>] [--hce]
  listen.py -h | --help

As the Target selected with the first positional argument listen.py
waits '--time T' seconds for activation by a remote device and prints
the local target configuration if not timed out. The listen period is
repeated after '--wait T' seconds if the '--repeat' flag is given.

Without the '--repeat' flag, the exit status is 0 when activated and 1
if timed out, with the '--repeat' flag it is 0 for termination by
keyboard interrupt (Ctrl-C). For argument errors and unsupported
targets listen.py exits with 2. If a local device is not found or was
removed listen.py exits with 3.

**Options:**

  -h, --help     show this help message and exit
  -t, --time T   listen time in seconds [default: 2.5]
  -w, --wait T   time between repetations [default: 1.0]
  -r, --repeat   repeat forever (cancel with Ctrl-C)
  -d, --debug    output debug log messages to stderr
  -v, --verbose  print and log more information
  --device PATH  local device search path [default: usb]
  --bitrate BR   set bitrate (default is 106 for A/B and 212 for F)
  --uid UID      tt2/tt4 identifier [default: 08010203]
  --idm IDM      tt3 identifier [default: 02FE010203040506]
  --pmm PMM      tt3 parameters [default: FFFFFFFFFFFFFFFF]
  --sys SYS      tt3 system code [default: 12FC]
  --id3 ID3      dep nfcid3 [default: 01FE0102030405060708]
  --gbt GBT      dep general bytes [default: 46666D010111]
  --hce          announce dep and tt4 support for Type A

**Examples:** ::

  listen.py tt2 --uid 08ABCDEF  # listen as Type 2 Tag with this UID
  listen.py tt3 --bitrate 424   # listen as Type 3 Tag at 424 kbps
  listen.py tt3 --sys 0003      # use the Suica system code for FeliCa
  listen.py dep --gbt ''        # send ATR response without general bytes
  listen.py dep --hce           # offer NFC-DEP Protocol and Type 4A Tag

"""
from __future__ import print_function

import os
import re
import sys
import struct
import time
import errno
import logging
```

(continues on next page)

(continued from previous page)

```
from binascii import hexlify

import nfc
import nfc.clf


def main(args):
    if args['--debug']:
        loglevel = logging.DEBUG - (1 if args['--verbose'] else 0)
        logging.getLogger("nfc.clf").setLevel(loglevel)
        logging.getLogger().setLevel(loglevel)

    try:
        try:
            waittime = float(args['--wait'])
        except ValueError:
            assert 0, "the '--wait T' argument must be a number"
        assert waittime >= 0, "the '--wait T' argument must be positive"
        try:
            timeout = float(args['--time'])
        except ValueError:
            assert 0, "the '--time T' argument must be a number"
        assert timeout >= 0, "the '--time T' argument must be positive"
    except AssertionError as error:
        print(str(error), file=sys.stderr)
        return 2

    try:
        clf = nfc.ContactlessFrontend(args['--device'])
    except IOError:
        print("no device found on path %r" % args['--device'], file=sys.stderr)
        return 3

    try:
        while True:
            target = None
            try:
                if args['tt2']:
                    target = listen_tta(timeout, clf, args)
                if args['tt3']:
                    target = listen_ttf(timeout, clf, args)
                if args['tt4']:
                    target = listen_tta(timeout, clf, args)
                if args['dep']:
                    target = listen_dep(timeout, clf, args)
                if target:
                    print("{0} {1}".format(time.strftime("%X"), target))
            except nfc.clf.CommunicationError as error:
                if args['--verbose']:
                    logging.error("%r", error)
            except AssertionError as error:
                print(str(error), file=sys.stderr)
                return 2

            if args['--repeat']:
                time.sleep(waittime)
            else:
```

(continues on next page)

(continued from previous page)

```
                return 0 if target is not None else 1

    except nfc.clf.UnsupportedTargetError as error:
        logging.error("%r", error)
        return 2

    except IOError as error:
        if error.errno != errno.EIO:
            logging.error("%r", error)
        else:
            logging.error("lost connection to local device")
        return 3

    except KeyboardInterrupt:
        pass

    finally:
        clf.close()


def listen_tta(timeout, clf, args):
    try:
        bitrate = (int(args['--bitrate']) if args['--bitrate'] else 106)
    except ValueError:
        assert 0, "the '--bitrate' argument must be an integer"
    assert bitrate >= 0, "the '--bitrate' argument must be a positive integer"

    try:
        uid = bytearray.fromhex(args['--uid'])
    except ValueError:
        assert 0, "the '--uid' argument must be hexadecimal"
    assert len(uid) in (4, 7, 10), "the '--uid' must be 4, 7, or 10 bytes"

    target = nfc.clf.LocalTarget(str(bitrate) + 'A')
    target.sens_res = bytearray(b"\x01\x01")
    target.sdd_res = uid
    target.sel_res = bytearray(b"\x00" if args['tt2'] else b"\x20")

    target = clf.listen(target, timeout)

    if target and target.tt2_cmd:
        logging.debug("rcvd TT2_CMD %s", hexlify(target.tt2_cmd).decode())

        # Verify that we can send a response.
        if target.tt2_cmd == b"\x30\x00":
            data = bytearray.fromhex("046FD536 11127A00 79C80000 E110060F")
        elif target.tt2_cmd[0] == 0x30:
            data = bytearray(16)
        else:
            logging.warning("communication not verified")
            return target

        try:
            clf.exchange(data, timeout=1)
            return target
        except nfc.clf.CommunicationError:
            logging.error("communication failure after activation")
```

(continues on next page)

(continued from previous page)

```
    if target and target.tt4_cmd:
        logging.debug("rcvd TT4_CMD %s", hexlify(target.tt4_cmd).decode())
        logging.warning("communication not verified")
        return target


def listen_ttf(timeout, clf, args):
    try:
        bitrate = (int(args['--bitrate']) if args['--bitrate'] else 212)
    except ValueError:
        assert 0, "the '--bitrate' argument must be an integer"
    assert bitrate >= 0, "the '--bitrate' argument must be a positive integer"

    try:
        idm = bytearray.fromhex(args['--idm'][0:16])
    except ValueError:
        assert 0, "the '--idm' argument must be hexadecimal"
    idm += os.urandom(8 - len(idm))

    try:
        pmm = bytearray.fromhex(args['--pmm'][0:16])
    except ValueError:
        assert 0, "the '--pmm' argument must be hexadecimal"
    pmm += (8 - len(pmm)) * b"\xFF"

    try:
        _sys = bytearray.fromhex(args['--sys'][0:4])
    except ValueError:
        assert 0, "the '--sys' argument must be hexadecimal"
    _sys += (2 - len(_sys)) * b"\xFF"

    target = nfc.clf.LocalTarget(str(bitrate) + 'F')
    target.sensf_res = b"\x01" + idm + pmm + _sys

    target = clf.listen(target, timeout)

    if target and target.tt3_cmd:
        if target.tt3_cmd[0] == 0x06:
            response = struct.pack("B", 29) + b"\7" + idm + b"\0\0\1" + \
                       bytearray(16)
            clf.exchange(response, timeout=0)
        elif target.tt3_cmd[0] == 0x0C:
            response = struct.pack("B", 13) + b"\x0D" + idm + b"\x01" + _sys
        else:
            logging.warning("communication not verified")
            return target

        try:
            clf.exchange(response, timeout=1)
            return target
        except nfc.clf.CommunicationError:
            logging.error("communication failure after activation")


def listen_dep(timeout, clf, args):
    try:
```

(continues on next page)

(continued from previous page)

```
        id3 = bytearray.fromhex(args['--id3'][0:20])
    except ValueError:
        assert 0, "the '--id3' argument must be hexadecimal"
    id3 += os.urandom(10 - len(id3))

    try:
        gbt = bytearray.fromhex(args['--gbt'])
    except ValueError:
        assert 0, "the '--gbt' argument must be hexadecimal"

    target = nfc.clf.LocalTarget()
    target.sensf_res = bytearray.fromhex("01") + id3[0:8] + bytearray(10)
    target.sens_res = bytearray.fromhex("0101")
    target.sdd_res = bytearray.fromhex("08") + id3[-3:]
    target.sel_res = bytearray.fromhex("60" if args['--hce'] else "40")
    target.atr_res = b"\xD5\x01" + id3 + b"\0\0\0\x08" + (
        b"\x32" if gbt else b"\0") + gbt

    target = clf.listen(target, timeout)
    if target and target.dep_req:
        logging.debug("rcvd DEP_REQ %s", hexlify(target.dep_req).decode())

        # Verify that we can indeed send a response. Note that we do
        # not handle a DID, but nobody is sending them anyway. Further
        # note that target.dep_req is without the frame length byte
        # but exchange() works on frames and so it has to be added.
        if target.dep_req.startswith(b"\xD4\x06\x80"):
            # older phones start with attention
            dep_res = bytearray.fromhex("04 D5 07 80")
        elif target.dep_req.startswith(b"\xD4\x06\x00"):
            # newer phones send information packet
            dep_res = bytearray.fromhex("06 D5 07 00 00 00")
        else:
            logging.warning("communication not verified")
            return target

        logging.debug("send DEP_RES %s",
                      hexlify(memoryview(dep_res)[1:]).decode())
        try:
            data = clf.exchange(dep_res, timeout=1)
            assert data and data[0] == len(data)
        except (nfc.clf.CommunicationError, AssertionError):
            logging.error("communication failure after activation")
            return None

        logging.debug("rcvd DEP_REQ %s",
                      hexlify(memoryview(data)[1:]).decode())
        mode = "passive" if target.sens_res or target.sensf_res else "active"
        logging.debug("activated in %s communication mode", mode)
        return target


if __name__ == '__main__':
    logging.basicConfig(format='%(relativeCreated)d ms [%(name)s] %(message)s')

    try:
        from docopt import docopt
```

(continues on next page)

(continued from previous page)

```
    except ImportError:
        sys.exit("the 'docopt' module is needed to execute this program")

    # remove restructured text formatting before input to docopt
    usage = re.sub(r'(?<=\n)\*\*(\w+:)\*\*.*\n', r'\1', __doc__)
    sys.exit(main(docopt(usage)))
```

5.5 rfstate.py

Observe the state of an external RF field.

Usage:

```
rfstate.py [options]
```

This is a simple utility to observe when a remote device activates and deactivates the 13.56 MHz carrier frequency. For each state change a message is printed with timestamp, the transition and time elapsed since the previous state change. This only works with some devices based on PN53x and uses nfcpy internal interfaces.

Options:

-h, --help	show this help message and exit
-t, --time T	listen time in seconds [default: 2.5]
-d, --debug	output debug log messages to stderr
-v, --verbose	print and log more information
--device PATH	local device search path [default: usb]

Source:

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-
# -----------------------------------------------------------------------------
# Copyright 2015 Stephen Tiedemann <stephen.tiedemann@gmail.com>
#
# Licensed under the EUPL, Version 1.1 or - as soon they
# will be approved by the European Commission - subsequent
# versions of the EUPL (the "Licence");
# You may not use this work except in compliance with the
# Licence.
# You may obtain a copy of the Licence at:
#
# https://joinup.ec.europa.eu/software/page/eupl
#
# Unless required by applicable law or agreed to in
# writing, software distributed under the Licence is
# distributed on an "AS IS" basis,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
# express or implied.
# See the Licence for the specific language governing
# permissions and limitations under the Licence.
# -----------------------------------------------------------------------------
"""Observe the state of an external RF field.
```

(continues on next page)

(continued from previous page)

```
**Usage:** ::

  rfstate.py [options]

This is a simple utility to observe when a remote device activates and
deactivates the 13.56 MHz carrier frequency. For each state change a
message is printed with timestamp, the transition and time elapsed
since the previous state change. This only works with some devices
based on PN53x and uses nfcpy internal interfaces.

**Options:**

  -h, --help     show this help message and exit
  -t, --time T   listen time in seconds [default: 2.5]
  -d, --debug    output debug log messages to stderr
  -v, --verbose  print and log more information
  --device PATH  local device search path [default: usb]

"""
from __future__ import print_function

import re
import sys
import time
import errno
import logging

import nfc
import nfc.clf
import nfc.clf.pn53x


def main(args):
    if args["--debug"]:
        loglevel = logging.DEBUG - (1 if args["--verbose"] else 0)
        logging.getLogger("nfc.clf").setLevel(loglevel)

    try:
        time_to_return = time.time() + float(args['--time'])
    except ValueError as e:
        logging.error("while parsing '--time' " + str(e))
        sys.exit(-1)

    clf = nfc.ContactlessFrontend()
    if clf.open(args['--device']):
        try:
            assert isinstance(clf.device, nfc.clf.pn53x.Device), \
                "rfstate.py does only work with PN53x based devices"
            chipset = clf.device.chipset

            regs = [("CIU_FIFOLevel", 0b10000000)]  # clear fifo
            regs.extend(zip(25 * ["CIU_FIFOData"], bytearray(25)))
            regs.extend([
                ("CIU_Command", 0b00000001),  # Configure command
                ("CIU_Control", 0b00000000),  # act as target (b4=0)
                ("CIU_TxControl", 0b10000000),  # disable output on TX1/TX2
                ("CIU_TxAuto", 0b00100000),  # wake up when rf level detected
```

(continues on next page)

(continued from previous page)

```
                ("CIU_CommIRq", 0b01111111),  # clear interrupt request bits
                ("CIU_DivIRq", 0b01111111),  # clear interrupt request bits
            ])
            chipset.write_register(*regs)

            if args["--verbose"]:
                time_t0 = time.time()
                chipset.read_register("CIU_Status1", "CIU_Status2")
                delta_t = time.time() - time_t0
                print("approx. %d samples/s" % int(1 / delta_t))

            status = chipset.read_register("CIU_Status1", "CIU_Status2")
            rfstate = "ON" if status[1] & 0b00100000 else "OFF"
            time_t0 = time.time()
            print("%.6f RF %s" % (time_t0, rfstate))

            while time.time() < time_to_return:
                status = chipset.read_register("CIU_Status1", "CIU_Status2")
                if rfstate == "OFF" and status[1] & 0x20 == 0x20:
                    rfstate = "ON"
                    time_t1 = time.time()
                    delta_t = time_t1 - time_t0
                    print("%.6f RF ON  after %.6f" % (time_t1, delta_t))
                    time_t0 = time_t1
                if rfstate == "ON" and status[1] & 0x20 == 0x00:
                    rfstate = "OFF"
                    time_t1 = time.time()
                    delta_t = time_t1 - time_t0
                    print("%.6f RF OFF after %.6f" % (time_t1, delta_t))
                    time_t0 = time_t1
        except nfc.clf.UnsupportedTargetError as error:
            print(repr(error))
        except IOError as error:
            if error.errno == errno.EIO:
                print("lost connection to local device")
            else:
                print(repr(error))
        except (NotImplementedError, AssertionError) as error:
            print(str(error))
        except KeyboardInterrupt:
            pass
        finally:
            clf.close()


if __name__ == '__main__':
    logging.basicConfig(format='%(relativeCreated)d ms [%(name)s] %(message)s')

    try:
        from docopt import docopt
    except ImportError:
        sys.exit("the 'docopt' module is needed to execute this program")

    # remove restructured text formatting before input to docopt
    usage = re.sub(r'(?<=\n)\*\*(\w+:)\*\*.*\n', r'\1', __doc__)
    sys.exit(main(docopt(usage)))
```

CHAPTER 6

Interoperability Tests

6.1 Logical Link Control Protocol

6.1.1 llcp-test-server.py

The LLCP test server program implements an NFC device that provides three distinct server applications:

1. A **connection-less echo server** that accepts connection-less transport mode PDUs. Service data units may have any size between zero and the maximum information unit size announced with the LLCP Link MIU parameter. Inbound service data units enter a linear buffer of service data units. The buffer has a capacity of two service data units. The first service data unit entering the buffer starts a delay timer of 2 seconds (echo delay). Expiration of the delay timer causes service data units in the buffer to be sent back to the original sender, which may be different for each service data unit, until the buffer is completely emptied. The buffer empty condition then re-enables the delay timer start event for the next service data unit.

2. A **connection-mode echo server** that waits for a connect request and then accepts and processes connection-oriented transport mode PDUs. Further connect requests will be rejected until termination of the data link connection. When accepting the connect request, the receive window parameter is transmitted with a value of 2.

   The connection-oriented mode echo service stores inbound service data units in a linear buffer of service data units. The buffer has a capacity of three service data units. The first service data unit entering the buffer starts a delay timer of 2 seconds (echo delay). Expiration of the delay timer causes service data units in the buffer to be sent back to the orignal sender until the buffer is completely emptied. The buffer empty condition then re-enables the delay timer start event for the next service data unit.

   The echo service determines itself as busy if it is unable to accept further incoming service data units.

3. A **connection-mode dump server** that accepts connections and then accepts and forgets all data received on a data link connection. This is mostly useful to measure transfer speed under load conditions.

Usage

```
$ llcp-test-server.py [-h|--help] [OPTION]...
```

Options

--loop, -l
Repeat the command endlessly, use Control-C to abort.

--mode {t,i}
Restrict the choice of NFC-DEP connection setup role to either `Target` (only listen) or `Initiator` (only poll). If this option is not given the dafault is to alternate between both roles with a randomized listen time.

--miu INT
Set a specific value for the LLCP Link MIU. The dafault value is 2175 octets.

--lto INT
Set a specific LLCP Link Timeout value. The default link timeout is 500 milliseconds.

--listen-time INT
Set the time to listen for initialization command from an NFC-DEP Initiator. The default listen time is 250 milliseconds.

--no-aggregation
Disable outbound packet aggregation for LLCP, i.e. do not generate LLCP AGF PDUs if multiple packets are waiting to be send. This is mostly to achieve communication with some older/buggy implementations.

-q
Do not print log messages except for errors and warnings.

-d MODULE
Output debug messages for MODULE to the log facility. Logs are written to <stderr> unless a log file is set with `-f`. MODULE is a string that corresponds to an *nfcpy* module or individual file, with dots between path components. For example, `-d nfc` enables all *nfcpy* debug logs, `-d nfc.tag` enables debug logs for all tag types, and `-d nfc.tag.tt3` enables debug logs only for type 3 tags. This option may be given multiple times to enable debug logs for several modules.

-f LOGFILE
Write debug log messages to <LOGFILE> instead of <stderr>. Info, warning and error logs will still be printed to <stderr> unless `-q` is set to supress info messages on <stderr>.

--nolog-symm
When operating in peer mode this option prevents logging of LLCP Symmetry PDUs from the `nfc.llcp.llc` module. Symmetry PDUs are exchanged regularly and quite frequently over an LLCP Link and are logged by default if debug output is enabled for the llcp module.

--device PATH
Use a specific reader or search only for a subset of readers. The syntax for PATH is:

- `usb[:vendor[:product]]` with optional *vendor* and *product* as four digit hexadecimal numbers, like `usb:054c:06c3` would open the first Sony RC-S380 reader and `usb:054c` the first Sony reader.
- `usb[:bus[:device]]` with optional *bus* and *device* number as three-digit decimal numbers, like `usb:001:023` would specifically mean the usb device with bus number 1 and device id 23 whereas `usb:001` would mean to use the first available reader on bus number 1.
- `tty:port:driver` with mandatory *port* and *driver* name should be used on Posix systems to open the serial port at device node `/dev/tty<port>` and load the driver from module `nfc/dev/<driver>.py`. A typical example would be `tty:USB0:arygon` for the Arygon APPx/ADRx at `/dev/ttyUSB0`.
- `com:port:driver` with mandatory *port* and *driver* name should be used on Windows systems to open the serial port `COM<port>` and load the `nfc/dev/<driver>.py` driver module.
- `udp[:host][:port]` with optional *host* name or address and *port* number will use a fake communication channel over UDP/IP. Either value may be omitted in which case *host* defaults to 'localhost' and *port* defaults to 54321.

6.1.2 llcp-test-client.py

Usage

```
$ llcp-test-client.py [-h|--help] [OPTION]...
```

Options

-t N, **--test** N
: Run test number *N*. May be set more than once.

-T, **--test-all**
: Run all tests.

--cl-echo SAP
: Service access point address of the connection-less mode echo server.

--co-echo SAP
: Service access point address of the connection-oriented mode echo server.

--loop, **-l**
: Repeat the command endlessly, use Control-C to abort.

--mode {t,i}
: Restrict the choice of NFC-DEP connection setup role to either `Target` (only listen) or `Initiator` (only poll). If this option is not given the dafault is to alternate between both roles with a randomized listen time.

--miu INT
: Set a specific value for the LLCP Link MIU. The dafault value is 2175 octets.

--lto INT
: Set a specific LLCP Link Timeout value. The default link timeout is 500 milliseconds.

--listen-time INT
: Set the time to listen for initialization command from an NFC-DEP Initiator. The default listen time is 250 milliseconds.

--no-aggregation
: Disable outbound packet aggregation for LLCP, i.e. do not generate LLCP AGF PDUs if multiple packets are waiting to be send. This is mostly to achieve communication with some older/buggy implementations.

-q
: Do not print log messages except for errors and warnings.

-d MODULE
: Output debug messages for MODULE to the log facility. Logs are written to <stderr> unless a log file is set with `-f`. MODULE is a string that corresponds to an *nfcpy* module or individual file, with dots between path components. For example, `-d nfc` enables all *nfcpy* debug logs, `-d nfc.tag` enables debug logs for all tag types, and `-d nfc.tag.tt3` enables debug logs only for type 3 tags. This option may be given multiple times to enable debug logs for several modules.

-f LOGFILE
: Write debug log messages to <LOGFILE> instead of <stderr>. Info, warning and error logs will still be printed to <stderr> unless `-q` is set to supress info messages on <stderr>.

--nolog-symm
: When operating in peer mode this option prevents logging of LLCP Symmetry PDUs from the `nfc.llcp.llc` module. Symmetry PDUs are exchanged regularly and quite frequently over an LLCP Link and are logged by default if debug output is enabled for the llcp module.

--device PATH
: Use a specific reader or search only for a subset of readers. The syntax for PATH is:

- `usb[:vendor[:product]]` with optional *vendor* and *product* as four digit hexadecimal numbers, like `usb:054c:06c3` would open the first Sony RC-S380 reader and `usb:054c` the first Sony reader.
- `usb[:bus[:device]]` with optional *bus* and *device* number as three-digit decimal numbers, like `usb:001:023` would specifically mean the usb device with bus number 1 and device id 23 whereas `usb:001` would mean to use the first available reader on bus number 1.
- `tty:port:driver` with mandatory *port* and *driver* name should be used on Posix systems to open the serial port at device node `/dev/tty<port>` and load the driver from module `nfc/dev/<driver>.py`. A typical example would be `tty:USB0:arygon` for the Arygon APPx/ADRx at `/dev/ttyUSB0`.
- `com:port:driver` with mandatory *port* and *driver* name should be used on Windows systems to open the serial port `COM<port>` and load the `nfc/dev/<driver>.py` driver module.
- `udp[:host][:port]` with optional *host* name or address and *port* number will use a fake communication channel over UDP/IP. Either value may be omitted in which case *host* defaults to 'localhost' and *port* defaults to 54321.

Test Scenarios

Link activation, symmetry and deactivation

```
$ llcp-test-client.py -t 1
```

Verify that the LLCP Link can be activated successfully, that the symmetry procedure is performed and the link can be intentionally deactivated.

1. Start the MAC link activation procedure on two implementations and verify that the version number parameter is received and version number agreement is achieved.
2. Verify for a duration of 5 seconds that SYMM PDUs are exchanged within the Link Timout values provided by the implementations.
3. Perform intentional link deactivation by sending a DISC PDU to the remote Link Management component. Verify that SYMM PDUs are no longer exchanged.

Connection-less information transfer

```
$ llcp-test-client.py -t 2
```

Verify that the source and destination access point address fields are correctly interpreted, the content of the information field is extracted as the service data unit and the service data unit can take any length between zero and the announced Link MIU. The LLCP Link must be activated prior to running this scenario and the Link MIU of the peer implementation must have been determined. In this scenario, sending of a service data unit (SDU) means that the SDU is carried within the information field of a UI PDU.

1. Send a service data unit of 128 octets length to the connection-less mode echo service and verify that the same SDU is sent back after the echo delay time.
2. Send within echo delay time with a time interval of at least 0.5 second two consecutive service data units of 128 octets length to the connection-less mode echo service and verify that both SDUs are sent back correctly.
3. Send within echo delay time with a time interval of at least 0.5 second three consecutive service data units of 128 octets length to the connection-less mode echo service and verify that the first two SDUs are sent back correctly and the third SDU is discarded.

4. Send a service data unit of zero octets length to the connection-less mode echo service and verify that the same zero length SDU is sent back after the echo delay time.
5. Send a service data unit of maximum octets length to the connection-less mode echo service and verify that the same SDU is sent back after the echo delay time. Note that the maximum length here must be the smaller value of both implementations Link MIU.

Connection-oriented information transfer

```
$ llcp-test-client.py -t 3
```

Verify that a data link connection can be established, a service data unit is received and sent back correctly and the data link connection can be terminated. The LLCP Link must be activated prior to running this scenario and the connection-oriented mode echo service must be in the unconnected state. In this scenario, sending of a service data unit (SDU) means that the SDU is carried within the information field of an I PDU.

1. Send a CONNECT PDU to the connection-oriented mode echo service and verify that the connection request is acknowledged with a CC PDU. The CONNECT PDU shall encode the RW parameter with a value of 2. Verify that the CC PDU encodes the RW parameter with a value of 2 (as specified for the echo server).
2. Send a single service data unit of 128 octets length over the data link connection and verify that the echo service sends an RR PDU before returning the same SDU after the echo delay time.
3. Send a DISC PDU to terminate the data link connection and verify that the echo service responds with a correct DM PDU.

Send and receive sequence number handling

```
$ llcp-test-client.py -t 4
```

Verify that a sequence of service data units that causes the send and receive sequence numbers to take all possible values is received and sent back correctly. The LLCP Link must be activated prior to running this scenario and the connection-oriented mode echo service must be in the unconnected state. In this scenario, sending of a service data unit (SDU) means that the SDU is carried within the information field of an I PDU.

1. Send a CONNECT PDU to the connection-oriented mode echo service and verify that the connection request is acknowledged with a CC PDU. The CONNECT PDU shall encode the RW parameter with a value of 2. Verify that the CC PDU encodes the RW parameter with a value of 2 (as specified for the echo server).
2. Send a sequence of at least 16 data units of each 128 octets length over the data link connection and verify that all SDUs are sent back correctly.
3. Send a DISC PDU to terminate the data link connection and verify that the echo service responds with a correct DM PDU.

Handling of receiver busy condition

```
$ llcp-test-client.py -t 5
```

Verify the handling of a busy condition. The LLCP Link must be activated prior to running this scenario and the connection-oriented mode echo service must be in the unconnected state. In this scenario, sending of a service data unit (SDU) shall mean that the SDU is carried within the information field of an I PDU.

1. Send a CONNECT PDU to the connection-oriented mode echo service and verify that the connect request is acknowledged with a CC PDU. The CONNECT PDU shall encode the RW parameter with a value of 0. Verify that the CC PDU encodes the RW parameter with a value of 2 (as specified for the echo server).
2. Send four service data units of 128 octets length over the data link connection and verify that the echo service enters the busy state when acknowledging the last packet.
3. Send a DISC PDU to terminate the data link connection and verify that the echo service responds with a correct DM PDU.

Rejection of connect request

```
$ llcp-test-client.py -t 6
```

Verify that an attempt to establish a second connection with the connection-oriented mode echo service is rejected. The LLCP Link must be activated prior to running this scenario.

1. Send a first CONNECT PDU to the connection-oriented mode echo service and verify that the connect request is acknowledged with a CC PDU.
2. Send a second CONNECT PDU to the connection-oriented mode echo service and verify that the connect request is rejected with a DM PDU and appropriate reason code.
3. Send a service data unit of 128 octets length over the data link connection and verify that the echo service returns the same SDU after the echo delay time.
4. Send a DISC PDU to terminate the data link connection and verify that the echo service responds with a correct DM PDU.

Connect by service name

```
$ llcp-test-client.py -t 7
```

Verify that a data link connection can be established by specifying a service name. The LLCP Link must be activated prior to running this scenario and the connection-oriented mode echo service must be in the unconnected state.

1. Send a CONNECT PDU with an SN parameter that encodes the value "urn:nfc:sn:co-echo" to the service discovery service access point address and verify that the connect request is acknowledged with a CC PDU.
2. Send a service data unit over the data link connection and verify that it is sent back correctly.
3. Send a DISC PDU to terminate the data link connection and verify that the echo service responds with a correct DM PDU.

Aggregation and disaggregation

```
$ llcp-test-client.py -t 8
```

Verify that the aggregation procedure is performed correctly. The LLCP Link must be activated prior to running this scenario. In this scenario, sending of a service data unit (SDU) shall mean that the SDU is carried within the information field of a UI PDU.

1. Send two service data units of 50 octets length to the connection-less mode echo service such that the two resulting UI PDUs will be aggregated into a single AGF PDU by the LLC sublayer. Verify that both SDUs are sent back correctly and in the same order.

2. Send three service data units of 50 octets length to the connection-less mode echo service such that the three resulting UI PDUs will be aggregated into a single AGF PDU by the LLC sublayer. Verify that the two first SDUs are sent back correctly and the third SDU is discarded.

Service name lookup

```
$ llcp-test-client.py -t 9
```

Verify that a service name is correctly resolved into a service access point address by the remote LLC. The LLCP Link must be activated prior to running this scenario. In this scenario, sending of a service data unit (SDU) shall mean that the SDU is carried within the information field of a UI PDU.

1. Send an SNL PDU with an SDREQ parameter in the information field that encodes the value "urn:nfc:sn:sdp" to the service discovery service access point address and verify that the request is responded with an SNL PDU that contains an SDRES parameter with the SAP value '1' and a TID value that is the same as the value encoded in the antecedently transmitted SDREQ parameter.
2. Send an SNL PDU with an SDREQ parameter in the information field that encodes the value "urn:nfc:sn:cl-echo" to the service discovery service access point address and verify that the request is responded with an SNL PDU that contains an SDRES parameter with a SAP value other than '0' and a TID value that is the same as the value encoded in the antecedently transmitted SDREQ parameter.
3. Send a service data unit of 128 octets length to the service access point address received in step 2 and verify that the same SDU is sent back after the echo delay time.
4. Send an SNL PDU with an SDREQ parameter in the information field that encodes the value "urn:nfc:sn:sdp-test" to the service discovery service access point address and verify that the request is responded with an SNL PDU that contains an SDRES parameter with the SAP value '0' and a TID value that is the same as the value encoded in the antecedently transmitted SDREQ parameter.

Send more data than allowed

```
$ llcp-test-client.py -t 10
```

Use invalid send sequence number

```
$ llcp-test-client.py -t 11
```

Use maximum data size on data link connection

```
$ llcp-test-client.py -t 12
```

Connect, release and connect again

```
$ llcp-test-client.py -t 13
```

Connect to unknown service name

```
$ llcp-test-client.py -t 14
```

Verify that a data link connection can be established by specifying a service name. The LLCP Link must be activated prior to running this scenario and the connection-oriented mode echo service must be in the unconnected state.

1. Send a CONNECT PDU with an SN parameter that encodes the value "urn:nfc:sn:co-echo-unknown" to the service discovery service access point address and verify that the connect request is rejected.

6.2 Simple NDEF Exchange Protocol

6.2.1 snep-test-server.py

The SNEP test server program implements an NFC device that provides two SNEP servers:

1. A **Default SNEP Server** that is compliant with the NFC Forum Default SNEP Server defined in section 6 of the SNEP specification.
2. A **Validation SNEP Server** that accepts SNEP Put and Get requests. A Put request causes the server to store the NDEF message transmitted with the request. A Get request causes the server to attempt to return a previously stored NDEF message of the same NDEF message type and identifier as transmitted with the request. The server will keep any number of distinct NDEF messages received with Put request until the client terminates the data link connection.

   The Validation SNEP Server uses the service name `urn:nfc:xsn:nfc-forum.org:snep-validation`, assigned for the purpose of validating the SNEP candidate specification prior to adoption.

Usage

```
$ snep-test-server.py [-h|--help] [OPTION]...
```

Options

`--loop, -l`
Repeat the command endlessly, use Control-C to abort.

`--mode` `{t,i}`
Restrict the choice of NFC-DEP connection setup role to either `Target` (only listen) or `Initiator` (only poll). If this option is not given the dafault is to alternate between both roles with a randomized listen time.

`--miu` `INT`
Set a specific value for the LLCP Link MIU. The dafault value is 2175 octets.

`--lto` `INT`
Set a specific LLCP Link Timeout value. The default link timeout is 500 milliseconds.

`--listen-time` `INT`
Set the time to listen for initialization command from an NFC-DEP Initiator. The default listen time is 250 milliseconds.

`--no-aggregation`
Disable outbound packet aggregation for LLCP, i.e. do not generate LLCP AGF PDUs if multiple packets are waiting to be send. This is mostly to achieve communication with some older/buggy implementations.

`-q`
Do not print log messages except for errors and warnings.

-d MODULE
: Output debug messages for MODULE to the log facility. Logs are written to <stderr> unless a log file is set with `-f`. MODULE is a string that corresponds to an *nfcpy* module or individual file, with dots between path components. For example, `-d nfc` enables all *nfcpy* debug logs, `-d nfc.tag` enables debug logs for all tag types, and `-d nfc.tag.tt3` enables debug logs only for type 3 tags. This option may be given multiple times to enable debug logs for several modules.

-f LOGFILE
: Write debug log messages to <LOGFILE> instead of <stderr>. Info, warning and error logs will still be printed to <stderr> unless `-q` is set to supress info messages on <stderr>.

--nolog-symm
: When operating in peer mode this option prevents logging of LLCP Symmetry PDUs from the `nfc.llcp.llc` module. Symmetry PDUs are exchanged regularly and quite frequently over an LLCP Link and are logged by default if debug output is enabled for the llcp module.

--device PATH
: Use a specific reader or search only for a subset of readers. The syntax for PATH is:

  - `usb[:vendor[:product]]` with optional *vendor* and *product* as four digit hexadecimal numbers, like `usb:054c:06c3` would open the first Sony RC-S380 reader and `usb:054c` the first Sony reader.
  - `usb[:bus[:device]]` with optional *bus* and *device* number as three-digit decimal numbers, like `usb:001:023` would specifically mean the usb device with bus number 1 and device id 23 whereas `usb:001` would mean to use the first available reader on bus number 1.
  - `tty:port:driver` with mandatory *port* and *driver* name should be used on Posix systems to open the serial port at device node `/dev/tty<port>` and load the driver from module `nfc/dev/<driver>.py`. A typical example would be `tty:USB0:arygon` for the Arygon APPx/ADRx at `/dev/ttyUSB0`.
  - `com:port:driver` with mandatory *port* and *driver* name should be used on Windows systems to open the serial port `COM<port>` and load the `nfc/dev/<driver>.py` driver module.
  - `udp[:host][:port]` with optional *host* name or address and *port* number will use a fake communication channel over UDP/IP. Either value may be omitted in which case *host* defaults to 'localhost' and *port* defaults to 54321.

6.2.2 snep-test-client.py

Usage

```
$ snep-test-client.py [-h|--help] [OPTION]...
```

Options

-t N, **--test** N
: Run test number *N*. May be set more than once.

-T, **--test-all**
: Run all tests.

--loop, **-l**
: Repeat the command endlessly, use Control-C to abort.

--mode {t,i}
: Restrict the choice of NFC-DEP connection setup role to either `Target` (only listen) or `Initiator` (only poll). If this option is not given the dafault is to alternate between both roles with a randomized listen time.

--miu INT
: Set a specific value for the LLCP Link MIU. The dafault value is 2175 octets.

--lto INT
Set a specific LLCP Link Timeout value. The default link timeout is 500 milliseconds.

--listen-time INT
Set the time to listen for initialization command from an NFC-DEP Initiator. The default listen time is 250 milliseconds.

--no-aggregation
Disable outbound packet aggregation for LLCP, i.e. do not generate LLCP AGF PDUs if multiple packets are waiting to be send. This is mostly to achieve communication with some older/buggy implementations.

-q
Do not print log messages except for errors and warnings.

-d MODULE
Output debug messages for MODULE to the log facility. Logs are written to <stderr> unless a log file is set with `-f`. MODULE is a string that corresponds to an *nfcpy* module or individual file, with dots between path components. For example, `-d nfc` enables all *nfcpy* debug logs, `-d nfc.tag` enables debug logs for all tag types, and `-d nfc.tag.tt3` enables debug logs only for type 3 tags. This option may be given multiple times to enable debug logs for several modules.

-f LOGFILE
Write debug log messages to <LOGFILE> instead of <stderr>. Info, warning and error logs will still be printed to <stderr> unless `-q` is set to supress info messages on <stderr>.

--nolog-symm
When operating in peer mode this option prevents logging of LLCP Symmetry PDUs from the `nfc.llcp.llc` module. Symmetry PDUs are exchanged regularly and quite frequently over an LLCP Link and are logged by default if debug output is enabled for the llcp module.

--device PATH
Use a specific reader or search only for a subset of readers. The syntax for PATH is:

- `usb[:vendor[:product]]` with optional *vendor* and *product* as four digit hexadecimal numbers, like `usb:054c:06c3` would open the first Sony RC-S380 reader and `usb:054c` the first Sony reader.
- `usb[:bus[:device]]` with optional *bus* and *device* number as three-digit decimal numbers, like `usb:001:023` would specifically mean the usb device with bus number 1 and device id 23 whereas `usb:001` would mean to use the first available reader on bus number 1.
- `tty:port:driver` with mandatory *port* and *driver* name should be used on Posix systems to open the serial port at device node `/dev/tty<port>` and load the driver from module `nfc/dev/<driver>.py`. A typical example would be `tty:USB0:arygon` for the Arygon APPx/ADRx at `/dev/ttyUSB0`.
- `com:port:driver` with mandatory *port* and *driver* name should be used on Windows systems to open the serial port `COM<port>` and load the `nfc/dev/<driver>.py` driver module.
- `udp[:host][:port]` with optional *host* name or address and *port* number will use a fake communication channel over UDP/IP. Either value may be omitted in which case *host* defaults to 'localhost' and *port* defaults to 54321.

Test Scenarios

Connect and terminate

```
$ snep-test-client.py -t 1
```

Verify that a data link connection with the remote validation server can be established and terminated gracefully and that the server returns to a connectable state.

1. Establish a data link connection with the Validation Server.
2. Verify that the data link connection was established successfully.
3. Close the data link connection with the Validation Server.
4. Establish a new data link connection with the Validation Server.
5. Verify that the data link connection was established successfully.
6. Close the data link connection with the Validation Server.

Unfragmented message exchange

```
$ snep-test-client.py -t 2
```

Verify that the remote validation server is able to receive unfragmented SNEP messages.

1. Establish a data link connection with the Validation Server.
2. Send a Put request with an NDEF message of no more than 122 octets total length.
3. Verify that the Validation Server accepted the Put request.
4. Send a Get request that identifies the NDEF message sent in step 2 to be retrieved.
5. Verify that the retrieved NDEF message is identical to the one transmitted in step 2.
6. Close the data link connection.

Fragmented message exchange

```
$ snep-test-client.py -t 3
```

Verify that the remote validation server is able to receive fragmented SNEP messages.

1. Establish a data link connection with the Validation Server.
2. Send a Put request with an NDEF message of more than 2170 octets total length.
3. Verify that the Validation Server accepted the Put request.
4. Send a Get request that identifies the NDEF message sent in step 2 to be retrieved.
5. Verify that the retrieved NDEF message is identical to the one transmitted in step 2.
6. Close the data link connection.

Multiple ndef messages

```
$ snep-test-client.py -t 4
```

Verify that the remote validation server accepts more than a single NDEF message on the same data link connection.

1. Establish a data link connection with the Validation Server.
2. Send a Put request with an NDEF message that differs from the NDEF message to be send in step 3.
3. Send a Put request with an NDEF message that differs from the NDEF message that has been send send in step 2.

4. Send a Get request that identifies the NDEF message sent in step 2 to be retrieved.
5. Send a Get request that identifies the NDEF message sent in step 3 to be retrieved.
6. Verify that the retrieved NDEF messages are identical to the NDEF messages transmitted in steps 2 and 3.
7. Close the data link connection.

Undeliverable resource

```
$ snep-test-client.py -t 5
```

Verify verify that the remote validation server responds appropriately if the client requests an NDEF message that exceeds the maximum acceptable length specified by the request.

1. Establish a data link connection with the Validation Server.
2. Send a Put request with an NDEF message of total lenght N.
3. Verify that the Validation Server accepted the Put request.
4. Send a Get request with the maximum acceptable lenght field set to $N \ 1$ and an NDEF message that identifies the NDEF message sent in step 2 to be retrieved.
5. Verify that the server replies with the appropriate response message.
6. Close the data link connection.

Unavailable resource

```
$ snep-test-client.py -t 6
```

Verify that the remote validation server responds appropriately if the client requests an NDEF message that is not available.

1. Establish a data link connection with the Validation Server.
2. Send a Get request that identifies an arbitrary NDEF message to be retrieved.
3. Verify that the server replies with the appropriate response message.
4. Close the data link connection.

Default server limits

```
$ snep-test-client.py -t 7
```

Verify verify that the remote default server accepts a Put request with an information field of up to 1024 octets, and that it rejects a Get request.

1. Establish a data link connection with the Default Server.
2. Send a Put request with an NDEF message of up to 1024 octets total length.
3. Verify that the Default Server replies with a Success response message.
4. Send a Get request with an NDEF message of arbitrary type and identifier.
5. Verify that the Default Server replies with a Not Implemented response message.

6. Close the data link connection.

6.3 Connection Handover

The **handover-test-server.py** and **handover-test-client.py** programs provide a test facility for the NFC Forum Connection Handover 1.2 specification.

6.3.1 handover-test-server.py

Usage:

```
$ handover-test-server.py [-h|--help] [OPTION]... [CARRIER]...
```

The handover test server implements the handover selector role. A handover client can connect to the server with the well-known service name `urn:nfc:sn:handover` and send handover request messages. The server replies with handover select messages populated with carriers provided through *CARRIER* arguments and matching the a carrier in the received handover request carrier list.

Each *CARRIER* argument must provide an NDEF message file, which may be a handover select message with one or more alternative carriers (including auxiliary data) or an alternative carrier record optionally followed by one or more auxiliary data records. Note that only the handover select message format allows to specify the carrier power state. All carriers including power state information and auxiliary data records are accumulated into a list of selectable carriers, ordered by argument position and carrier sequence within a handover select message.

Unless the `--skip-local` option is given, the server attempts to include carriers that are locally available on the host device. Local carriers are always added after all *CARRIER* arguments.

Note: Local carrier detection currently requires a Linux OS with the bluez Bluetooth stack and D-Bus. This is true for many Linux distributions, but has so far only be tested on Ubuntu.

Options:

`--skip-local`
Skip the local carrier detection. Without this option the handover test server tries to discover locally available carriers and consider them in the selection process. Local carriers are considered after all carriers provided manually.

`--select` `NUM`
Return at most *NUM* carriers with the handover select message. The default is to return all matching carriers.

`--delay` `INT`
Delay the handover response for the number of milliseconds specified as INT. The handover specification says that the server should answer within 1 second and if it doesn't the client may assume a processing error.

`--recv-miu` `INT`
Set the maximum information unit size for inbound LLCP packets on the data link connection between the server and the remote client. This value is transmitted with the CC PDU to the remote client.

`--recv-buf` `INT`
Set the receive window size for inbound LLCP packets on the data link connection between the server and the remote client. This value is transmitted with the CC PDU to the remote client.

`--quirks`
This option causes the handover test server to try support non-compliant implementations if possible and as known. Currently implemented work-arounds are:

- a 'urn:nfc:sn:snep' server is enabled and accepts the GET request with a handover request message that was implemented in Android Jelly Bean
- the version of the handover request message sent by Android Jelly Bean is changed to 1.1 to accomodate the missing collision resolution record that is required for version 1.2.
- the incorrect type-name-format encoding in handover carrier records sent by some Sony Xperia phones is corrected to mime-type.

Test Scenarios

Empty handover select response

```
$ handover-test-server.py --select 0
```

Verify that the remote handover client accepts a handover select message that has no alternative carriers.

A carrier that is being activated

```
$ ndeftool.py make btcfg 01:02:03:04:05:06 --activating | handover-test-server --skip-
↪local -
```

Verify that the remote handover client understands and tries to connect to a Bluetooth carrier that is in the process of activation.

Delayed handover select response

```
$ examples/handover-test-server.py --delay 10000
```

Check hot the remote handover implementation behaves if the handover select response is delayed for about 10 seconds. This test intends to help identify user interface issues.

6.3.2 handover-test-client.py

Usage

```
$ handover-test-client.py [-h|--help] [OPTION]... [CARRIER]...
```

The handover test client implements the handover requester role. The handover client connects to the remote server with well-known service name `urn:nfc:sn:handover` and sends handover request messages populated with carriers provided through one or more *CARRIER* arguments or implicitly if tests from the test suite are executed. The client expects the server to reply with handover select messages that list carriers matching one or more of the carriers sent with the handover request carrier list.

Each *CARRIER* argument must provide an NDEF message file which may be a handover message with one or more alternative carriers (including auxiliary data) or an alternative carrier record followed by zero or more auxiliary data records. Note that only the handover message format allows to specify the carrier power state. All carriers, including power state information and auxiliary data records, are accumulated into a list of requestable carriers ordered by argument position and carrier sequence within a handover message.

Options

-t N, **--test** N
Run test number *N* from the test suite. Multiple tests can be specified.

--relax
The `--relax` option affects how missing optional, but highly recommended, handover data is handled when running test scenarios. Without `--relax` any missing data is regarded as a test error that terminates test execution. With the `--relax` option set only a warning message is logged.

--recv-miu INT
Set the maximum information unit size for inbound LLCP packets on the data link connection between the client and the remote server. This value is transmitted with the CONNECT PDU to the remote server.

--recv-buf INT
Set the receive window size for inbound LLCP packets on the data link connection between the client and the remote server. This value is transmitted with the CONNECT PDU to the remote server.

--quirks
This option causes the handover test client to try support non-compliant implementations as much as possible, including and beyond the `--relax` behavor. The modifications activated with `--quirks` are:

- After test procedures are completed the client does not terminate the LLCP link but waits until the link is disrupted to prevent the NFC stack segfault and recovery on pre 4.1 Android devices.
- Try sending the handover request message with a SNEP GET request to the remote default SNEP server if the `urn:nfc:sn:handover` service is not available.

Test Scenarios

Presence and connectivity

```
$ handover-test-client.py -t 1
```

Verify that the remote device has the connection handover service active and that the client can open, close and re-open a connection with the server.

1. Connect to the remote handover service.
2. Close the data link conection.
3. Connect to the remote handover service.
4. Close the data link conection.

Empty carrier list

```
$ handover-test-client.py -t 2
```

Verify that the handover server responds to a handover request without alternative carriers with a handover select message that also has no alternative carriers.

1. Connect to the remote handover service.
2. Send a handover request message containing zero alternative carriers.
3. Verify that the server returns a handover select message within no more than 3 seconds; and that the message contains zero alternative carriers.
4. Close the data link conection.

Version handling

```
$ handover-test-client.py -t 3
```

Verify that the remote handover server handles historic and future handover request version numbers. This test is run as a series of steps where for each step the connection to the server is established and closed after completion. For all steps the configuration sent is a Bluetooth carrier for device address `01:02:03:04:05:06`.

1. Connect to the remote handover service.
2. Send a handover request message with version `1.2`.
3. Verify that the server replies with version `1.2`.
4. Close the data link conection.
5. Connect to the remote handover service.
6. Send a handover request message with version `1.1`.
7. Verify that the server replies with version `1.2`.
8. Close the data link conection.
9. Connect to the remote handover service.
10. Send a handover request message with version `1.15`.
11. Verify that the server replies with version `1.2`.
12. Close the data link conection.
13. Connect to the remote handover service.
14. Send a handover request message with version `15.0`.
15. Verify that the server replies with version `1.2`.
16. Close the data link conection.

Bluetooth just-works pairing

```
$ handover-test-client.py -t 4
```

Verify that the `application/vnd.bluetooth.ep.oob` alternative carrier is correctly evaluated and replied. This test is only applicable if the peer device does have Bluetooth connectivity.

1. Connect to the remote handover service.
2. Send a handover request message with a single alternative carrier of type `application/vnd.bluetooth.ep.oob` and power state `active`. Secure pairing hash and randomizer are not provided with the Bluetooth configuration.
3. Verify that the server returns a handover select message within no more than 3 seconds; that the message contains exactly one alternative carrier with type `application/vnd.bluetooth.ep.oob` and power state `active` or `activating`; that the Bluetooth local device name is transmitted; and that secure simple pairing hash and randomizer are not transmitted. Issues a warning if class of device/service or service class UUID attributes are not transmitted.
4. Close the data link conection.

Bluetooth secure pairing

```
$ handover-test-client.py -t 5
```

Verify that the `application/vnd.bluetooth.ep.oob` alternative carrier is correctly evaluated and replied. This test is only applicable if the peer device does have Bluetooth connectivity.

1. Connect to the remote handover service.
2. Send a handover request message with a single alternative carrier of type `application/vnd.bluetooth.ep.oob` and power state `active`. Secure pairing hash and randomizer are transmitted with the Bluetooth configuration.
3. Verify that the server returns a handover select message within no more than 3 seconds; that the message contains exactly one alternative carrier with type `application/vnd.bluetooth.ep.oob` and power state `active` or `activating`; that the Bluetooth local device name is transmitted; and that secure simple pairing hash and randomizer are transmitted. Issues a warning if class of device/service or service class UUID attributes are not transmitted.
4. Close the data link conection.

Unknown carrier type

```
$ handover-test-client.py -t 6
```

Verify that the remote handover server returns a select message without alternative carriers if a single carrier of unknown type was sent with the handover request.

1. Connect to the remote handover service.
2. Send a handover request message with a single alternative carrier of type `urn:nfc:ext:nfcpy.org:unknown-carrier-type`.
3. Verify that the server returns a handover select message with an empty alternative carrier selection.
4. Close the data link conection.

Two handover requests

```
$ handover-test-client.py -t 7
```

Verify that the remote handover server does not close the data link connection after the first handover request message.

1. Connect to the remote handover service.
2. Send a handover request with a single carrier of unknown type
3. Send a handover request with a single Bluetooth carrier
4. Close the data link conection.

Reserved-future-use check

```
$ handover-test-client.py -t 8
```

Verify that reserved bits are set to zero and optional reserved bytes are not present in the payload of the alternative carrier record. This test requires that the remote server selects a Bluetooth alternative carrier if present in the request.

1. Connect to the remote handover service.
2. Send a handover request with a single Bluetooth carrier
3. Verify that an alternative carrier record is present; that reserved bits in the first octet are zero; and that the record payload ends with the last auxiliary data reference.
4. Close the data link conection.

Skip meaningless records

```
$ handover-test-client.py -t 9
```

Verify that records that have no defined meaning in the payload of a handover request record are ignored. This test assumes that the remote server selects a Bluetooth alternative carrier if present in the request.

1. Connect to the remote handover service.
2. Send a handover request with a single Bluetooth carrier and a meaningless text record as the first record of the handover request record payload.
3. Verify that an Bluetooth alternative carrier record is returned.
4. Close the data link conection.

6.4 Personal Health Device Communication

6.4.1 phdc-test-manager.py

This program implements an NFC device that provides a PHDC manager with the well-known service name `urn:nfc:sn:phdc` and a non-standard PHDC manager with the experimental service name `urn:nfc:xsn:nfc-forum.org:phdc-validation`.

Usage

```
$ phdc-test-manager.py [-h|--help] [OPTION]...
```

Options

`--loop, -l`
Repeat the command endlessly, use Control-C to abort.

`--mode` `{t,i}`
Restrict the choice of NFC-DEP connection setup role to either `Target` (only listen) or `Initiator` (only poll). If this option is not given the dafault is to alternate between both roles with a randomized listen time.

`--miu` `INT`
Set a specific value for the LLCP Link MIU. The dafault value is 2175 octets.

`--lto` `INT`
Set a specific LLCP Link Timeout value. The default link timeout is 500 milliseconds.

`--listen-time` `INT`
Set the time to listen for initialization command from an NFC-DEP Initiator. The default listen time is 250 milliseconds.

--no-aggregation
Disable outbound packet aggregation for LLCP, i.e. do not generate LLCP AGF PDUs if multiple packets are waiting to be send. This is mostly to achieve communication with some older/buggy implementations.

--wait
After reading or writing a tag, wait until it is removed before returning. This option is implicit when the option `--loop` is set.

--technology `{A,B,F}`
Poll only for tags of a specific technology. The technologies NFC-A, NFC-B, and NFC-F are defined in the NFC Forum Digital Specification. The technology indicator is case insensitive. The default is to poll for all technologies.

-q
Do not print log messages except for errors and warnings.

-d `MODULE`
Output debug messages for MODULE to the log facility. Logs are written to <stderr> unless a log file is set with `-f`. MODULE is a string that corresponds to an *nfcpy* module or individual file, with dots between path components. For example, `-d nfc` enables all *nfcpy* debug logs, `-d nfc.tag` enables debug logs for all tag types, and `-d nfc.tag.tt3` enables debug logs only for type 3 tags. This option may be given multiple times to enable debug logs for several modules.

-f `LOGFILE`
Write debug log messages to <LOGFILE> instead of <stderr>. Info, warning and error logs will still be printed to <stderr> unless `-q` is set to supress info messages on <stderr>.

--nolog-symm
When operating in peer mode this option prevents logging of LLCP Symmetry PDUs from the `nfc.llcp.llc` module. Symmetry PDUs are exchanged regularly and quite frequently over an LLCP Link and are logged by default if debug output is enabled for the llcp module.

--device `PATH`
Use a specific reader or search only for a subset of readers. The syntax for PATH is:

- `usb[:vendor[:product]]` with optional *vendor* and *product* as four digit hexadecimal numbers, like `usb:054c:06c3` would open the first Sony RC-S380 reader and `usb:054c` the first Sony reader.
- `usb[:bus[:device]]` with optional *bus* and *device* number as three-digit decimal numbers, like `usb:001:023` would specifically mean the usb device with bus number 1 and device id 23 whereas `usb:001` would mean to use the first available reader on bus number 1.
- `tty:port:driver` with mandatory *port* and *driver* name should be used on Posix systems to open the serial port at device node `/dev/tty<port>` and load the driver from module `nfc/dev/<driver>.py`. A typical example would be `tty:USB0:arygon` for the Arygon APPx/ADRx at `/dev/ttyUSB0`.
- `com:port:driver` with mandatory *port* and *driver* name should be used on Windows systems to open the serial port `COM<port>` and load the `nfc/dev/<driver>.py` driver module.
- `udp[:host][:port]` with optional *host* name or address and *port* number will use a fake communication channel over UDP/IP. Either value may be omitted in which case *host* defaults to 'localhost' and *port* defaults to 54321.

6.4.2 phdc-test-agent.py p2p

Usage

```
$ phdc-test-agent.py p2p [-h|--help] [OPTION]...
```

Options

-t N, **--test** N
Run test number *N*. May be set more than once.

-T, --test-all
Run all tests.

--loop, -l
Repeat the command endlessly, use Control-C to abort.

--mode {t,i}
Restrict the choice of NFC-DEP connection setup role to either `Target` (only listen) or `Initiator` (only poll). If this option is not given the dafault is to alternate between both roles with a randomized listen time.

--miu INT
Set a specific value for the LLCP Link MIU. The dafault value is 2175 octets.

--lto INT
Set a specific LLCP Link Timeout value. The default link timeout is 500 milliseconds.

--listen-time INT
Set the time to listen for initialization command from an NFC-DEP Initiator. The default listen time is 250 milliseconds.

--no-aggregation
Disable outbound packet aggregation for LLCP, i.e. do not generate LLCP AGF PDUs if multiple packets are waiting to be send. This is mostly to achieve communication with some older/buggy implementations.

-q
Do not print log messages except for errors and warnings.

-d MODULE
Output debug messages for MODULE to the log facility. Logs are written to <stderr> unless a log file is set with `-f`. MODULE is a string that corresponds to an *nfcpy* module or individual file, with dots between path components. For example, `-d nfc` enables all *nfcpy* debug logs, `-d nfc.tag` enables debug logs for all tag types, and `-d nfc.tag.tt3` enables debug logs only for type 3 tags. This option may be given multiple times to enable debug logs for several modules.

-f LOGFILE
Write debug log messages to <LOGFILE> instead of <stderr>. Info, warning and error logs will still be printed to <stderr> unless `-q` is set to supress info messages on <stderr>.

--nolog-symm
When operating in peer mode this option prevents logging of LLCP Symmetry PDUs from the `nfc.llcp.llc` module. Symmetry PDUs are exchanged regularly and quite frequently over an LLCP Link and are logged by default if debug output is enabled for the llcp module.

--device PATH
Use a specific reader or search only for a subset of readers. The syntax for PATH is:

- `usb[:vendor[:product]]` with optional *vendor* and *product* as four digit hexadecimal numbers, like `usb:054c:06c3` would open the first Sony RC-S380 reader and `usb:054c` the first Sony reader.
- `usb[:bus[:device]]` with optional *bus* and *device* number as three-digit decimal numbers, like `usb:001:023` would specifically mean the usb device with bus number 1 and device id 23 whereas `usb:001` would mean to use the first available reader on bus number 1.
- `tty:port:driver` with mandatory *port* and *driver* name should be used on Posix systems to open the serial port at device node `/dev/tty<port>` and load the driver from module `nfc/dev/<driver>.py`. A typical example would be `tty:USB0:arygon` for the Arygon APPx/ADRx at `/dev/ttyUSB0`.

- `com:port:driver` with mandatory *port* and *driver* name should be used on Windows systems to open the serial port `COM<port>` and load the `nfc/dev/<driver>.py` driver module.
- `udp[:host][:port]` with optional *host* name or address and *port* number will use a fake communication channel over UDP/IP. Either value may be omitted in which case *host* defaults to 'localhost' and *port* defaults to 54321.

Test Scenarios

Connect, Associate and Release

```
$ phdc-test-agent.py p2p -t 1
```

Verify that the Agent can connect to the PHDC Manager, associate with the IEEE Manager and finally release the association.

1. Establish communication distance between the Thermometer Peer Agent and the Manager device.
2. Connect to the `urn:nfc:sn:phdc` service.
3. Send a Thermometer Association Request.
4. Verify that the Manager sends a Thermometer Association Response.
5. Wait 3 seconds not sending any IEEE APDU, then send an Association Release Request.
6. Verify that the Manager sends an Association Release Response
7. Disconnect from the `urn:nfc:sn:phdc` service.
8. Move Agent and Manager device out of communication range.

Association after Release

```
$ phdc-test-agent.py p2p -t 2
```

Verify that the Agent can again associate with the Manager after a first association has been established and released.

1. Establish communication distance between the Thermometer Peer Agent and the Manager device.
2. Connect to the `urn:nfc:sn:phdc` service.
3. Send a Thermometer Association Request.
4. Verify that the Manager sends a Thermometer Association Response.
5. Disconnect from the `urn:nfc:sn:phdc` service.
6. Connect to the `urn:nfc:sn:phdc` service.
7. Send a Thermometer Association Request.
8. Verify that the Manager sends a Thermometer Association Response.
9. Send a Association Release Request.
10. Verify that the Manager sends a Association Release Response.
11. Disconnect from the `urn:nfc:sn:phdc` service.
12. Move Agent and Manager device out of communication range.

PHDC PDU Fragmentation and Reassembly

```
$ phdc-test-agent.py p2p -t 3
```

Verify that large PHDC PDUs are correctly fragmented and reassembled.

1. Establish communication distance between the Validation Agent and the Manager device.
2. Connect to the `urn:nfc:xsn:nfc-forum.org:phdc-validation` service.
3. Send a PHDC PDU with an Information field of 2176 random octets.
4. Verify to receive an PHDC PDU that contains the same random octets in reversed order.
5. Disconnect from the `urn:nfc:xsn:nfc-forum.org:phdc-validation` service.
6. Move Agent and Manager device out of communication range.

6.4.3 phdc-test-agent.py tag

Usage

```
$ phdc-test-agent.py tag [-h|--help] [OPTION]...
```

Options

`-t` `N`, `--test` `N`
Run test number *N*. May be set more than once.

`-T`, `--test-all`
Run all tests.

`--loop`, `-l`
Repeat the command endlessly, use Control-C to abort.

`-q`
Do not print log messages except for errors and warnings.

`-d` `MODULE`
Output debug messages for MODULE to the log facility. Logs are written to <stderr> unless a log file is set with `-f`. MODULE is a string that corresponds to an *nfcpy* module or individual file, with dots between path components. For example, `-d nfc` enables all *nfcpy* debug logs, `-d nfc.tag` enables debug logs for all tag types, and `-d nfc.tag.tt3` enables debug logs only for type 3 tags. This option may be given multiple times to enable debug logs for several modules.

`-f` `LOGFILE`
Write debug log messages to <LOGFILE> instead of <stderr>. Info, warning and error logs will still be printed to <stderr> unless `-q` is set to supress info messages on <stderr>.

`--nolog-symm`
When operating in peer mode this option prevents logging of LLCP Symmetry PDUs from the `nfc.llcp.llc` module. Symmetry PDUs are exchanged regularly and quite frequently over an LLCP Link and are logged by default if debug output is enabled for the llcp module.

`--device` `PATH`
Use a specific reader or search only for a subset of readers. The syntax for PATH is:

- `usb[:vendor[:product]]` with optional *vendor* and *product* as four digit hexadecimal numbers, like `usb:054c:06c3` would open the first Sony RC-S380 reader and `usb:054c` the first Sony reader.

- `usb[:bus[:device]]` with optional *bus* and *device* number as three-digit decimal numbers, like `usb:001:023` would specifically mean the usb device with bus number 1 and device id 23 whereas `usb:001` would mean to use the first available reader on bus number 1.
- `tty:port:driver` with mandatory *port* and *driver* name should be used on Posix systems to open the serial port at device node `/dev/tty<port>` and load the driver from module `nfc/dev/<driver>.py`. A typical example would be `tty:USB0:arygon` for the Arygon APPx/ADRx at `/dev/ttyUSB0`.
- `com:port:driver` with mandatory *port* and *driver* name should be used on Windows systems to open the serial port `COM<port>` and load the `nfc/dev/<driver>.py` driver module.
- `udp[:host][:port]` with optional *host* name or address and *port* number will use a fake communication channel over UDP/IP. Either value may be omitted in which case *host* defaults to 'localhost' and *port* defaults to 54321.

Test Scenarios

Discovery, Association and Release

```
$ phdc-test-agent.py tag -t 1
```

Verify that a PHDC Tag Agent is discovered by a PHDC Manager and IEEE APDU exchange is successful.

1. Establish communication distance between the Thermometer Tag Agent and the Manager.
2. Send a Thermometer Association Request.
3. Verify that the Manager sends a Thermometer Association Response.
4. Wait 3 seconds not sending any IEEE APDU, then send an Association Release Request.
5. Verify that the Manager sends a Association Release Response.
6. Move Thermometer Tag Agent and Manager out of communication range.

Association after Release

```
$ phdc-test-agent.py tag -t 2
```

Verify that a Tag Agent can again associate with the Manager after a first association has been established and released.

1. Establish communication distance between the Thermometer Tag Agent and the Manager.
2. Send a Thermometer Association Request.
3. Verify that the Manager sends a Thermometer Association Response.
4. Send an Association Release Request.
5. Verify that the Manager sends a Association Release Response.
6. Wait 3 seconds not sending any IEEE APDU, then send a Thermometer Association Request.
7. Verify that the Manager sends a Thermometer Association Response.
8. Move Thermometer Tag Agent and Manager out of communication range.

Activation with invalid settings

```
$ phdc-test-agent.py tag -t 3
```

Verify that a PHDC Manager refuses communication with a Tag Agent that presents an invalid PHDC record payload during activation.

1. Establish communication distance between the Tag Agent and the Manager.
2. Send the first PHDC PDU with invalid settings in one or any of the MC, LC or MD fields.
3. Verify that the Manager stops further PHDC communication with the Tag Agent.

Activation with invalid RFU value

```
$ phdc-test-agent.py tag -t 4
```

Verify that a PHDC Manager communicates with a Tag Agent that presents a PHDC record payload with an invalid RFU value during activation.

1. Establish communication distance between the Tag Agent and the Manager.
2. Send the first PHDC PDU with an invalid value in the RFU field.
3. Verify that the Manager continues PHDC communication with the Tag Agent.

6.5 Generate Test Tags

This page contains instructions to generate tags for testing reader compliance with NFC Forum Tag Type, NDEF and RTD specifications. The tools used are in the `examples` directory.

6.5.1 Type 3 Tags

Attribute Block Tests

This is a collection of tags to test processing of the the Type 3 Tag attribute information block. These can be used to verify if the NFC device correctly reads or writes tags with different attribute information, both valid and invalid. Below figure (from the NFC Forum Type 3 Tag Operation Specification) shows the Attribute Information Format.

User Block No.00															
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15
Ver	Nbr	Nbw	Nmaxb		unused	unused	unused	unused	WriteF	RW Flag	Ln			Checksum	

TT3_READ_BV_001

```
$ ./tagtool.py format
$ ./ndeftool.py make smartposter http://nfcpy.readthedocs.org/ -t "nfcpy␣
↪documentation hosted on readthedocs" | ./tagtool.py load -
$ ./tagtool.py format tt3 --len 80 --max 5 --rw 0
```

- Settings: Len = Nmaxb * 16, RWFlag = 0x00
- Expected: Fully used tag. Read all data stored (Len)

TT3_READ_BV_002

```
$ ./tagtool.py format
$ ./ndeftool.py make smartposter http://nfcpy.readthedocs.org/ -t "nfcpy documentation
↪" | ./tagtool.py load -
$ ./tagtool.py format tt3 --len 58 --rw 0 --nbr 1
```

- Settings: Nbr = 1, RWFlag = 0x00
- Expected: Identify as „Read Only“ (normal read-only tag, read only 1 block at a time)

TT3_READ_BV_003

```
$ ./tagtool.py format
$ ./ndeftool.py make smartposter http://nfcpy.readthedocs.org/ -t "nfcpy documentation
↪" | ./tagtool.py load -
$ ./tagtool.py format tt3 --len 58 --rw 0 --max 3
```

- Nbr > Nbmax, RWFlag = 0x00
- Read Nbmax blocks (NOT read Nbr blocks)

TT3_READ_BV_004

```
$ ./tagtool.py format
$ ./ndeftool.py make smartposter http://nfcpy.readthedocs.org/ -t "nfcpy documentation
↪" | ./tagtool.py load -
$ ./tagtool.py format tt3 --len 58 --rw 0 --wf 15
```

- WriteFlag = 0x0F, RWFlag = 0x00
- Identify as „corrupted data“ (previous write interrupted)

TT3_READ_BV_005

```
$ ./tagtool.py format
$ ./ndeftool.py make smartposter http://nfcpy.readthedocs.org/ -t "nfcpy documentation
↪" | ./tagtool.py load -
$ ./tagtool.py format tt3 --len 58 --rw 0 --max 3
```

- Nmaxb * 16 < Len, RWFlag = 0x00
- Identify as „Corrupted data“ (invalid length)

TT3_READ_BV_006

```
$ ./tagtool.py format
$ ./ndeftool.py make smartposter http://nfcpy.readthedocs.org/ -t `python -c
↪'print(810*"nfcpy")'` | ./tagtool.py load -
$ ./tagtool.py format tt3 --len 4495 --rw 0
```

- Nmaxb > 255, Len > 255, RWFlag = 0x00
- Read all data. Identify as „Read Only“. Write prohibited. (normal read-only tag)
- Requires a tag with more than 4 kbyte NDEF capacity

TT3_READ_BI_001

```
$ ./tagtool.py format
$ ./ndeftool.py make smartposter http://nfcpy.readthedocs.org/ -t "nfcpy documentation
↪" | ./tagtool.py load -
$ ./tagtool.py format tt3 --len 58 --rw 0 --nbr 0 --nbw 0
```

- Nbr = 0, Nbw = 0, RWFlag = 0x00
- Identify as „Corrupted data“ (invalid attribute information block)

TT3_READ_BI_002

```
$ ./tagtool.py format
$ ./ndeftool.py make smartposter http://nfcpy.readthedocs.org/ -t "nfcpy documentation
↪" | ./tagtool.py load -
$ ./tagtool.py format tt3 --len 58 --rw 0 --crc 4660
```

- Checksum invalid, RWFlag = 0x00
- Identify as „Corrupted data“ (invalid attribute information block)

TT3_READ_BI_003

```
$ ./tagtool.py format
$ ./ndeftool.py make smartposter http://nfcpy.readthedocs.org/ -t "nfcpy documentation
↪" | ./tagtool.py load -
$ ./tagtool.py format tt3 --len 58 --rw 0 --ver 2.0
```

- Version = 2.0, RWFlag = 0x00
- Identify as unknown version

TT3_READ_BI_004

```
$ ./tagtool.py format
$ ./ndeftool.py make smartposter http://nfcpy.readthedocs.org/ -t "nfcpy documentation
↪" | ./tagtool.py load -
$ ./tagtool.py format tt3 --len 58 --rw 0 --rfu 255
```

- All unused bytes in attribute block = 0xFF

- Ignore when reading RWFlag = 0x00

TT3_WRITE_BV_001

```
$ ./tagtool.py format tt3 --rw 0
```

- RWFlag = 0x00, no content
- Identify as „Read Only“. Write prohibited. (normal read-only tag)

TT3_WRITE_BV_002

```
$ ./tagtool.py format tt3 --rw 1
```

- RWFlag = 0x01, no content
- Identify as „Read/Write“. Write permitted. (normal writtable tag)

TT3_WRITE_BV_003

```
$ ./tagtool.py format tt3 --rw 0 --max 4
```

- Nbw > Nbmax, RWFlag = 0x01
- Write Nbmax blocks (**not** write Nbw blocks)

CHAPTER 7

Module Reference

7.1 nfc

7.1.1 nfc.ContactlessFrontend

class nfc.**ContactlessFrontend**
Shorthand for *nfc.clf.ContactlessFrontend*.

7.2 nfc.clf

class nfc.clf.**ContactlessFrontend**(*path=None*)

Bases: object

This class is the main interface for working with contactless devices. The *connect()* method provides easy access to the contactless functionality through automated discovery of remote cards and devices and activation of appropiate upper level protocols for further interaction. The *sense()*, *listen()* and *exchange()* methods provide a low-level interface for more specialized tasks.

An instance of the *ContactlessFrontend* class manages a single contactless device locally connect through either USB, TTY or COM port. A special UDP port driver allows for emulation of a contactless device that connects through UDP to another emulated contactless device for test and development of higher layer functions.

A locally connected contactless device can be opened by either supplying a *path* argument when an an instance of the contactless frontend class is created or by calling *open()* at a later time. In either case the *path* argument must be constructed as described in *open()* and the same exceptions may occur. The difference is that *open()* returns False if a device could not be found whereas the initialization method raises IOError with errno.ENODEV.

The methods of the *ContactlessFrontend* class are thread-safe.

open(*path*)

Open a contactless reader identified by the search *path*.

The *open()* method searches and then opens a contactless reader device for further communication. The *path* argument can be flexibly constructed to identify more or less precisely the device to open. A *path* that only partially identifies a device is completed by search. The first device that is found and successfully opened causes *open()* to return True. If no device is found return value is False. If a device was found but could not be opened then *open()* returns False if *path* was partial or raise IOError if *path* was fully qualified. Typical I/O error reasons are errno.EACCES if the calling process has insufficient access rights or errno.EBUSY if the device is used by another process.

A path is constructed as follows:

usb[:vendor[:product]]

with optional *vendor* and *product* as four digit hexadecimal numbers. For example, usb:054c:06c3 would open the first Sony RC-S380 reader while usb:054c would open the first Sony reader found on USB.

usb[:bus[:device]]

with optional *bus* and *device* number as three-digit decimals. For example, usb:001:023 would open the device enumerated as number 23 on bus 1 while usb:001 would open the first device found on bust 1. Note that a new device number is generated every time the device is plugged into USB. Bus and device numbers are shown by lsusb.

tty:port:driver

with mandatory *port* and *driver* name. This is for Posix systems to open the serial port /dev/tty<port> and use the driver module nfc/dev/<driver>.py for access. For example, tty:USB0:arygon would open /dev/ttyUSB0 and load the Arygon APPx/ADRx driver.

com:port:driver

with mandatory *port* and *driver* name. This is for Windows systems to open the serial port COM<port> and use the driver module nfc/dev/<driver>.py for access.

udp[:host][:port]

with optional *host* name or address and *port* number. This will emulate a communication channel over UDP/IP. The defaults for *host* and *port* are localhost:54321.

close()
Close the contacless reader device.

connect(**options*)
Connect with a Target or Initiator

The calling thread is blocked until a single activation and deactivation has completed or a callback function supplied as the keyword argument `terminate` returns a true value. The example below makes *connect()* return after 5 seconds, regardless of whether a peer device was connected or not.

```
>>> import nfc, time
>>> clf = nfc.ContactlessFrontend('usb')
>>> after5s = lambda: time.time() - started > 5
>>> started = time.time(); clf.connect(llcp={}, terminate=after5s)
```

Connect options are given as keyword arguments with dictionary values. Possible options are:

- `rdwr={key: value, ...}` - options for reader/writer
- `llcp={key: value, ...}` - options for peer to peer
- `card={key: value, ...}` - options for card emulation

Reader/Writer Options

'targets' [iterable] A list of bitrate and technology type strings that will produce the *RemoteTarget* objects to discover. The default is `('106A', '106B', '212F')`.

'on-startup' [function(targets)] This function is called before any attempt to discover a remote card. The *targets* argument provides a list of *RemoteTarget* objects prepared from the 'targets' bitrate and technology type strings. The function must return a list of of those *RemoteTarget* objects that shall be finally used for discovery, those targets may have additional attributes. An empty list or anything else that evaluates false will remove the 'rdwr' option completely.

'on-discover' [function(target)] This function is called when a *RemoteTarget* has been discovered. The *target* argument contains the technology type specific discovery responses and should be evaluated for multi-protocol support. The target will be further activated only if this function returns a true value. The default function depends on the 'llcp' option, if present then the function returns True only if the target does not indicate peer to peer protocol support, otherwise it returns True for all targets.

'on-connect' [function(tag)] This function is called when a remote tag has been activated. The *tag* argument is an instance of class *nfc.tag.Tag* and can be used for tag reading and writing within the callback or in a separate thread. Any true return value instructs *connect()* to wait until the tag is no longer present and then return True, any false return value implies immediate return of the *nfc.tag.Tag* object.

'on-release' [function(tag)] This function is called when the presence check was run (the 'on-connect' function returned a true value) and determined that communication with the *tag* has become impossible, or when the 'terminate' function returned a true value. The *tag* object may be used for cleanup actions but not for communication.

'iterations' [integer] This determines the number of sense cycles performed between calls to the terminate function. Each iteration searches once for all specified targets. The default value is 5 iterations and between each iteration is a waiting time determined by the 'interval' option described below. As an effect of math there will be no waiting time if iterations is set to 1.

'interval' [float] This determines the waiting time between iterations. The default value of 0.5 seconds is considered a sensible tradeoff between responsiveness in terms of tag discovery and power consumption. It should be clear that changing this value will impair one or the other. There is no free beer.

'beep-on-connect': boolean If the device supports beeping or flashing an LED, automatically perform this functionality when a tag is successfully detected AND the 'on-connect' function returns a true value. Defaults to True.

```
import nfc

def on_startup(targets):
    for target in targets:
        target.sensf_req = bytearray.fromhex("0012FC0000")
    return targets

def on_connect(tag):
    print(tag)

rdwr_options = {
    'targets': ['212F', '424F'],
    'on-startup': on_startup,
    'on-connect': on_connect,
}
with nfc.ContactlessFrontend('usb') as clf:
    tag = clf.connect(rdwr=rdwr_options)
    if tag.ndef:
        print(tag.ndef.message.pretty())
```

Peer To Peer Options

'on-startup' [function(llc)] This function is called before any attempt to establish peer to peer communication. The *llc* argument provides the *LogicalLinkController* that may be used to allocate and bind listen sockets for local services. The function should return the *llc* object if activation shall continue. Any other value removes the 'llcp' option.

'on-connect' [function(llc)] This function is called when peer to peer communication is successfully established. The *llc* argument provides the now activated *LogicalLinkController* ready for allocation of client communication sockets and data exchange in separate work threads. The function should a true value return more or less immediately, unless it wishes to handle the logical link controller run loop by itself and anytime later return a false value.

'on-release' [function(llc)] This function is called when the symmetry loop was run (the 'on-connect' function returned a true value) and determined that communication with the remote peer has become impossible, or when the 'terminate' function returned a true value. The *llc* object may be used for cleanup actions but not for communication.

'role' [string] This attribute determines whether the local device will restrict itself to either `'initiator'` or `'target'` mode of operation. As Initiator the local device will try to discover a remote device. As Target it waits for being discovered. The default is to alternate between both roles.

'miu' [integer] This attribute sets the maximum information unit size that is announced to the remote device during link activation. The default and also smallest possible value is 128 bytes.

'lto' [integer] This attribute sets the link timeout value (given in milliseconds) that is announced to the remote device during link activation. It informs the remote device that if the local device does not return a protocol data unit before the timeout expires, the communication link is broken and can not be recovered. The *lto* is an important part of the user experience, it ultimately tells when the user should no longer expect communication to continue. The default value is 500 millisecond.

'agf' [boolean] Some early phone implementations did not properly handle aggregated protocol data units. This attribute allows to disable the use af aggregation at the cost of efficiency. Aggregation is disabled with a false value. The default is to use aggregation.

'brs' [integer] For the local device in Initiator role the bit rate selector determines the the bitrate to negotiate with the remote Target. The value may be 0, 1, or 2 for 106, 212, or 424 kbps, respectively. The default is to negotiate 424 kbps.

'acm' [boolean] For the local device in Initiator role this attribute determines whether a remote Target may also be activated in active communication mode. In active communication mode both peer devices mutually generate a radio field when sending. The default is to use passive communication mode.

'rwt' [float] For the local device in Target role this attribute sets the response waiting time announced during link activation. The response waiting time is a medium access layer (NFC-DEP) value that indicates when the remote Initiator shall attempt error recovery after missing a Target response. The value is the waiting time index *wt* that determines the effective response waiting time by the formula `rwt = 4096/13.56E6 * pow(2, wt)`. The value shall not be greater than 14. The default value is 8 and yields an effective response waiting time of 77.33 ms.

'lri' [integer] For the local device in Initiator role this attribute sets the length reduction for medium access layer (NFC-DEP) information frames. The value may be 0, 1, 2, or 3 for a maximum payload size of 64, 128, 192, or 254 bytes, respectively. The default value is 3.

'lrt' [integer] For the local device in Target role this attribute sets the length reduction for medium access layer (NFC-DEP) information frames. The value may be 0, 1, 2, or 3 for a maximum payload size of 64, 128, 192, or 254 bytes, respectively. The default value is 3.

```
import nfc
import nfc.llcp
import threading

def server(socket):
    message, address = socket.recvfrom()
    socket.sendto("It's me!", address)
    socket.close()

def client(socket):
    socket.sendto("Hi there!", address=32)
    socket.close()

def on_startup(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.LOGICAL_DATA_LINK)
    socket.bind(address=32)
    threading.Thread(target=server, args=(socket,)).start()
    return llc

def on_connect(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.LOGICAL_DATA_LINK)
    threading.Thread(target=client, args=(socket,)).start()
    return True

llcp_options = {
    'on-startup': on_startup,
    'on-connect': on_connect,
}
with nfc.ContactlessFrontend('usb') as clf:
    clf.connect(llcp=llcp_options)
    print("link terminated")
```

Card Emulation Options

'on-startup' [function(target)] This function is called to prepare a local target for discovery. The input argument is a fresh instance of an unspecific *LocalTarget* that can be set to the desired bitrate

and modulation type and populated with the type specific discovery responses (see *listen()* for response data that is needed). The fully specified target object must then be returned.

'on-discover' [function(target)] This function is called when the *LocalTarget* has been discovered. The *target* argument contains the technology type specific discovery commands. The target will be further activated only if this function returns a true value. The default function always returns True.

'on-connect' [function(tag)] This function is called when the local target was discovered and a *nfc.tag.TagEmulation* object successfully initialized. The function receives the emulated *tag* object which stores the first command received after inialization as `tag.cmd`. The function should return a true value if the tag.process_command() and tag.send_response() methods shall be called repeatedly until either the remote device terminates communication or the 'terminate' function returns a true value. The function should return a false value if the *connect()* method shall return immediately with the emulated *tag* object.

'on-release' [function(tag)] This function is called when the Target was released by the Initiator or simply moved away, or if the terminate callback function has returned a true value. The emulated *tag* object may be used for cleanup actions but not for communication.

```
import nfc

def on_startup(target):
    idm = bytearray.fromhex("01010501b00ac30b")
    pmm = bytearray.fromhex("03014b024f4993ff")
    sys = bytearray.fromhex("1234")
    target.brty = "212F"
    target.sensf_res = chr(1) + idm + pmm + sys
    return target

def on_connect(tag):
    print("discovered by remote reader")
    return True

def on_release(tag):
    print("remote reader is gone")
    return True

card_options = {
    'on-startup': on_startup,
    'on-connect': on_connect,
    'on-release': on_release,
}
with nfc.ContactlessFrontend('usb') as clf:
    clf.connect(card=card_options)
```

Return Value

The *connect()* method returns `None` if there were no options left after the 'on-startup' functions have been executed or when the 'terminate' function returned a true value. It returns `False` when terminated by any of the following exceptions: `KeyboardInterrupt`, `IOError`, *UnsupportedTargetError*.

The *connect()* method returns a *Tag*, *LogicalLinkController*, or *TagEmulation* object if the associated 'on-connect' function returned a false value to indicate that it will handle presence check, peer to peer symmetry loop, or command/response processing by itself.

sense(**targets*, ***options*)

Discover a contactless card or listening device.

Note: The *sense()* method is intended for experts with a good understanding of the commands and

responses exchanged during target activation (the notion used for commands and responses follows the NFC Forum Digital Specification). If the greater level of control is not needed it is recommended to use the *connect()* method.

All positional arguments build the list of potential *targets* to discover and must be of type *RemoteTarget*. Keyword argument *options* may be the number of `iterations` of the sense loop set by *targets* and the `interval` between iterations. The return value is either a *RemoteTarget* instance or `None`.

```
>>> import nfc, nfc.clf
>>> from binascii import hexlify
>>> clf = nfc.ContactlessFrontend("usb")
>>> target1 = nfc.clf.RemoteTarget("106A")
>>> target2 = nfc.clf.RemoteTarget("212F")
>>> print(clf.sense(target1, target2, iterations=5, interval=0.2))
106A(sdd_res=04497622D93881, sel_res=00, sens_res=4400)
```

A **Type A Target** is specified with the technology letter `A` following the bitrate to be used for the SENS_REQ command (almost always must the bitrate be 106 kbps). To discover only a specific Type A target, the NFCID1 (UID) can be set with a 4, 7, or 10 byte `sel_req` attribute (cascade tags are handled internally).

```
>>> target = nfc.clf.RemoteTarget("106A")
>>> print(clf.sense(target))
106A sdd_res=04497622D93881 sel_res=00 sens_res=4400
>>> target.sel_req = bytearray.fromhex("04497622D93881")
>>> print(clf.sense(target))
106A sdd_res=04497622D93881 sel_res=00 sens_res=4400
>>> target.sel_req = bytearray.fromhex("04497622")
>>> print(clf.sense(target))
None
```

A **Type B Target** is specified with the technology letter `B` following the bitrate to be used for the SENSB_REQ command (almost always must the bitrate be 106 kbps). A specific application family identifier can be set with the first byte of a `sensb_req` attribute (the second byte PARAM is ignored when it can not be set to local device, 00h is a safe value in all cases).

```
>>> target = nfc.clf.RemoteTarget("106B")
>>> print(clf.sense(target))
106B sens_res=50E5DD3DC900000011008185
>>> target.sensb_req = bytearray.fromhex("0000")
>>> print(clf.sense(target))
106B sens_res=50E5DD3DC900000011008185
>>> target.sensb_req = bytearray.fromhex("FF00")
>>> print(clf.sense(target))
None
```

A **Type F Target** is specified with the technology letter `F` following the bitrate to be used for the SENSF_REQ command (the typically supported bitrates are 212 and 424 kbps). The default SENSF_REQ command allows all targets to answer, requests system code information, and selects a single time slot for the SENSF_RES response. This can be changed with the `sensf_req` attribute.

```
>>> target = nfc.clf.RemoteTarget("212F")
>>> print(clf.sense(target))
212F sensf_res=0101010601B00ADE0B03014B024F4993FF12FC
>>> target.sensf_req = bytearray.fromhex("0012FC0000")
```

(continues on next page)

(continued from previous page)

```
>>> print(clf.sense(target))
212F sensf_res=0101010601B00ADE0B03014B024F4993FF
>>> target.sensf_req = bytearray.fromhex("00ABCD0000")
>>> print(clf.sense(target))
None
```

An **Active Communication Mode P2P Target** search is selected with an `atr_req` attribute. The choice of bitrate and modulation type is 106A, 212F, and 424F.

```
>>> atr = bytearray.fromhex("D4000102030405060708091000000030")
>>> target = clf.sense(nfc.clf.RemoteTarget("106A", atr_req=atr))
>>> if target and target.atr_res:
>>>     print(hexlify(target.atr_res).decode())
d501c023cae6b3182afe3dee0000000e3246666d01011103020013040196
>>> target = clf.sense(nfc.clf.RemoteTarget("424F", atr_req=atr))
>>> if target and target.atr_res:
>>>     print(hexlify(target.atr_res).decode())
d501dc0104f04584e15769700000000e3246666d01011103020013040196
```

Some drivers must modify the ATR_REQ to cope with hardware limitations, for example change length reduction value to reduce the maximum size of target responses. The ATR_REQ that has been send is given by the `atr_req` attribute of the returned RemoteTarget object.

A **Passive Communication Mode P2P Target** responds to 106A discovery with bit 6 of SEL_RES set to 1, and to 212F/424F discovery (when the request code RC is 0 in the SENSF_REQ command) with an NFCID2 that starts with 01FEh in the SENSF_RES response. Responses below are from a Nexus 5 configured for NFC-DEP Protocol (SEL_RES bit 6 is set) and Type 4A Tag (SEL_RES bit 5 is set).

```
>>> print(clf.sense(nfc.clf.RemoteTarget("106A")))
106A sdd_res=08796BEB sel_res=60 sens_res=0400
>>> sensf_req = bytearray.fromhex("00FFFF0000")
>>> print(clf.sense(nfc.clf.RemoteTarget("424F", sensf_req=sensf_req)))
424F sensf_res=0101FE1444EFB88FD50000000000000000
```

Errors found in the *targets* argument list raise exceptions only if exactly one target is given. If multiple targets are provided, any target that is not supported or has invalid attributes is just ignored (but is logged as a debug message).

Exceptions

- `IOError` (ENODEV) when a local contacless communication device has not been opened or communication with the local device is no longer possible.
- *`nfc.clf.UnsupportedTargetError`* if the single target supplied as input is not supported by the active driver. This exception is never raised when *`sense()`* is called with multiple targets, those unsupported are then silently ignored.

`listen`(*target*, *timeout*)

Listen *timeout* seconds to become activated as *target*.

Note: The *`listen()`* method is intended for experts with a good understanding of the commands and responses exchanged during target activation (the notion used for commands and responses follows the NFC Forum Digital Specification). If the greater level of control is not needed it is recommended to use the *`connect()`* method.

The *target* argument is a *`LocalTarget`* object that provides bitrate, technology type and response

data attributes. The return value is either a *LocalTarget* object with bitrate, technology type and request/response data attributes or None.

An **P2P Target** is selected when the atr_res attribute is set. The bitrate and technology type are decided by the Initiator and do not need to be specified. The sens_res, sdd_res and sel_res attributes for Type A technology as well as the sensf_res attribute for Type F technolgy must all be set.

When activated, the bitrate and type are set to the current communication values, the atr_req attribute contains the ATR_REQ received from the Initiator and the dep_req attribute contains the first DEP_REQ received after activation. If the Initiator has changed communication parameters, the psl_req attribute holds the PSL_REQ that was received. The atr_res (and the psl_res if transmitted) are also made available.

If the local target was activated in passive communication mode either the Type A response (sens_res, sdd_res, sel_res) or Type F response (sensf_res) attributes will be present.

With a Nexus 5 on a reader connected via USB the following code should be working and produce similar output (the Nexus 5 prioritizes active communication mode):

```
>>> import nfc, nfc.clf
>>> clf = nfc.ContactlessFrontend("usb")
>>> atr_res = "d50101fe0102030405060708000000083246666d010110"
>>> target = nfc.clf.LocalTarget()
>>> target.sensf_res = bytearray.fromhex("0101FE"+16*"FF")
>>> target.sens_res = bytearray.fromhex("0101")
>>> target.sdd_res = bytearray.fromhex("08010203")
>>> target.sel_res = bytearray.fromhex("40")
>>> target.atr_res = bytearray.fromhex(atr_res)
>>> print(clf.listen(target, timeout=2.5))
424F atr_res=D50101FE0102030405060708000000083246666D010110 ...
```

A **Type A Target** is selected when atr_res is not present and the technology type is A. The bitrate should be set to 106 kbps, even if a driver supports higher bitrates they would need to be set after activation. The sens_res, sdd_res and sel_res attributes must all be provided.

```
>>> target = nfc.clf.Localtarget("106A")
>>> target.sens_res = bytearray.fromhex("0101"))
>>> target.sdd_res = bytearray.fromhex("08010203")
>>> target.sel_res = bytearray.fromhex("00")
>>> print(clf.listen(target, timeout=2.5))
106A sdd_res=08010203 sel_res=00 sens_res=0101 tt2_cmd=3000
```

A **Type B Target** is selected when atr_res is not present and the technology type is B. Unfortunately none of the supported devices supports Type B technology for listen and an *nfc.clf.UnsupportedTargetError* exception will be the only result.

```
>>> target = nfc.clf.LocalTarget("106B")
>>> try: clf.listen(target, 2.5)
... except nfc.clf.UnsupportedTargetError: print("sorry")
...
sorry
```

A **Type F Target** is selected when atr_res is not present and the technology type is F. The bitrate may be 212 or 424 kbps. The sensf_res attribute must be provided.

```
>>> idm, pmm, sys = "02FE010203040506", "FFFFFFFFFFFFFFFF", "12FC"
>>> target = nfc.clf.LocalTarget("212F")
>>> target.sensf_res = bytearray.fromhex("01" + idm + pmm + sys)
```

(continues on next page)

(continued from previous page)

```
>>> print(clf.listen(target, 2.5))
212F sensf_req=00FFFF0003 tt3_cmd=0C02FE010203040506 ...
```

Exceptions

- `IOError` (ENODEV) when a local contacless communication device has not been opened or communication with the local device is no longer possible.
- *`nfc.clf.UnsupportedTargetError`* if the single target supplied as input is not supported by the active driver. This exception is never raised when *`sense()`* is called with multiple targets, those unsupported are then silently ignored.

exchange (*send_data*, *timeout*)
Exchange data with an activated target (*send_data* is a command frame) or as an activated target (*send_data* is a response frame). Returns a target response frame (if data is send to an activated target) or a next command frame (if data is send from an activated target). Returns None if the communication link broke during exchange (if data is sent as a target). The timeout is the number of seconds to wait for data to return, if the timeout expires an nfc.clf.TimeoutException is raised. Other nfc.clf.CommunicationError exceptions may be raised if an error is detected during communication.

max_send_data_size
The maximum number of octets that can be send with the *`exchange()`* method in the established operating mode.

max_recv_data_size
The maximum number of octets that can be received with the *`exchange()`* method in the established operating mode.

class nfc.clf.**RemoteTarget**(*brty*, ***kwargs*)
Bases: `nfc.clf.Target`

A RemoteTarget instance provides bitrate and technology type and command/response data of a remote card or device that, when input to `sense()`, shall be attempted to discover and, when returned by `sense()`, has been discovered by the local device. Command/response data attributes, whatever name, default to None.

brty
A string that combines bitrate and technology type, e.g. '106A'.

class nfc.clf.**LocalTarget**(*brty='106A'*, ***kwargs*)
Bases: `nfc.clf.Target`

A LocalTarget instance provides bitrate and technology type and command/response data of the local card or device that, when input to `listen()`, shall be made available for discovery and, when returned by `listen()`, has been discovered by a remote device. Command/response data attributes, whatever name, default to None.

brty
A string that combines bitrate and technology type, e.g. '106A'.

exception nfc.clf.**Error**
Bases: `Exception`

Base class for exceptions specific to the contacless frontend module.

- UnsupportedTargetError
- CommunicationError
  - ProtocolError
  - TransmissionError
  - TimeoutError

- BrokenLinkError

exception nfc.clf.**UnsupportedTargetError**
Bases: *nfc.clf.Error*

The *RemoteTarget* input to *ContactlessFrontend.sense()* or *LocalTarget* input to *ContactlessFrontend.listen()* is not supported by the local device.

exception nfc.clf.**CommunicationError**
Bases: *nfc.clf.Error*

Base class for communication errors.

exception nfc.clf.**ProtocolError**
Bases: *nfc.clf.CommunicationError*

Raised when an NFC Forum Digital Specification protocol error occured.

exception nfc.clf.**TransmissionError**
Bases: *nfc.clf.CommunicationError*

Raised when an NFC Forum Digital Specification transmission error occured.

exception nfc.clf.**TimeoutError**
Bases: *nfc.clf.CommunicationError*

Raised when an NFC Forum Digital Specification timeout error occured.

exception nfc.clf.**BrokenLinkError**
Bases: *nfc.clf.CommunicationError*

The remote device (Reader/Writer or P2P Device) has deactivated the RF field or is no longer within communication distance.

7.2.1 Contactless Frontend

Note: The contactless frontend defined in this module is also available as *nfc.ContactlessFrontend*.

class nfc.clf.**ContactlessFrontend**(*path=None*)
Bases: object

This class is the main interface for working with contactless devices. The *connect()* method provides easy access to the contactless functionality through automated discovery of remote cards and devices and activation of appropiate upper level protocols for further interaction. The *sense()*, *listen()* and *exchange()* methods provide a low-level interface for more specialized tasks.

An instance of the *ContactlessFrontend* class manages a single contactless device locally connect through either USB, TTY or COM port. A special UDP port driver allows for emulation of a contactless device that connects through UDP to another emulated contactless device for test and development of higher layer functions.

A locally connected contactless device can be opened by either supplying a *path* argument when an an instance of the contactless frontend class is created or by calling *open()* at a later time. In either case the *path* argument must be constructed as described in *open()* and the same exceptions may occur. The difference is that *open()* returns False if a device could not be found whereas the initialization method raises IOError with errno.ENODEV.

The methods of the *ContactlessFrontend* class are thread-safe.

open(*path*)
Open a contactless reader identified by the search *path*.

The *open()* method searches and then opens a contactless reader device for further communication. The *path* argument can be flexibly constructed to identify more or less precisely the device to open. A *path* that only partially identifies a device is completed by search. The first device that is found and successfully opened causes *open()* to return True. If no device is found return value is False. If a device was found but could not be opened then *open()* returns False if *path* was partial or raise `IOError` if *path* was fully qualified. Typical I/O error reasons are `errno.EACCES` if the calling process has insufficient access rights or `errno.EBUSY` if the device is used by another process.

A path is constructed as follows:

`usb[:vendor[:product]]`

with optional *vendor* and *product* as four digit hexadecimal numbers. For example, `usb:054c:06c3` would open the first Sony RC-S380 reader while `usb:054c` would open the first Sony reader found on USB.

`usb[:bus[:device]]`

with optional *bus* and *device* number as three-digit decimals. For example, `usb:001:023` would open the device enumerated as number 23 on bus 1 while `usb:001` would open the first device found on bust 1. Note that a new device number is generated every time the device is plugged into USB. Bus and device numbers are shown by `lsusb`.

`tty:port:driver`

with mandatory *port* and *driver* name. This is for Posix systems to open the serial port `/dev/tty<port>` and use the driver module `nfc/dev/<driver>.py` for access. For example, `tty:USB0:arygon` would open `/dev/ttyUSB0` and load the Arygon APPx/ADRx driver.

`com:port:driver`

with mandatory *port* and *driver* name. This is for Windows systems to open the serial port `COM<port>` and use the driver module `nfc/dev/<driver>.py` for access.

`udp[:host][:port]`

with optional *host* name or address and *port* number. This will emulate a communication channel over UDP/IP. The defaults for *host* and *port* are `localhost:54321`.

close()
Close the contacless reader device.

connect(***options*)
Connect with a Target or Initiator

The calling thread is blocked until a single activation and deactivation has completed or a callback function supplied as the keyword argument `terminate` returns a true value. The example below makes *connect()* return after 5 seconds, regardless of whether a peer device was connected or not.

```
>>> import nfc, time
>>> clf = nfc.ContactlessFrontend('usb')
>>> after5s = lambda: time.time() - started > 5
>>> started = time.time(); clf.connect(llcp={}, terminate=after5s)
```

Connect options are given as keyword arguments with dictionary values. Possible options are:

- `rdwr={key: value, ...}` - options for reader/writer
- `llcp={key: value, ...}` - options for peer to peer

- `card={key: value, ...}` - options for card emulation

Reader/Writer Options

'targets' [iterable] A list of bitrate and technology type strings that will produce the *RemoteTarget* objects to discover. The default is `('106A', '106B', '212F')`.

'on-startup' [function(targets)] This function is called before any attempt to discover a remote card. The *targets* argument provides a list of *RemoteTarget* objects prepared from the 'targets' bitrate and technology type strings. The function must return a list of of those *RemoteTarget* objects that shall be finally used for discovery, those targets may have additional attributes. An empty list or anything else that evaluates false will remove the 'rdwr' option completely.

'on-discover' [function(target)] This function is called when a *RemoteTarget* has been discovered. The *target* argument contains the technology type specific discovery responses and should be evaluated for multi-protocol support. The target will be further activated only if this function returns a true value. The default function depends on the 'llcp' option, if present then the function returns True only if the target does not indicate peer to peer protocol support, otherwise it returns True for all targets.

'on-connect' [function(tag)] This function is called when a remote tag has been activated. The *tag* argument is an instance of class *nfc.tag.Tag* and can be used for tag reading and writing within the callback or in a separate thread. Any true return value instructs *connect()* to wait until the tag is no longer present and then return True, any false return value implies immediate return of the *nfc.tag.Tag* object.

'on-release' [function(tag)] This function is called when the presence check was run (the 'on-connect' function returned a true value) and determined that communication with the *tag* has become impossible, or when the 'terminate' function returned a true value. The *tag* object may be used for cleanup actions but not for communication.

'iterations' [integer] This determines the number of sense cycles performed between calls to the terminate function. Each iteration searches once for all specified targets. The default value is 5 iterations and between each iteration is a waiting time determined by the 'interval' option described below. As an effect of math there will be no waiting time if iterations is set to 1.

'interval' [float] This determines the waiting time between iterations. The default value of 0.5 seconds is considered a sensible tradeoff between responsiveness in terms of tag discovery and power consumption. It should be clear that changing this value will impair one or the other. There is no free beer.

'beep-on-connect': boolean If the device supports beeping or flashing an LED, automatically perform this functionality when a tag is successfully detected AND the 'on-connect' function returns a true value. Defaults to True.

```
import nfc

def on_startup(targets):
    for target in targets:
        target.sensf_req = bytearray.fromhex("0012FC0000")
    return targets

def on_connect(tag):
    print(tag)

rdwr_options = {
    'targets': ['212F', '424F'],
    'on-startup': on_startup,
    'on-connect': on_connect,
}
```

(continues on next page)

(continued from previous page)

```
with nfc.ContactlessFrontend('usb') as clf:
    tag = clf.connect(rdwr=rdwr_options)
    if tag.ndef:
        print(tag.ndef.message.pretty())
```

Peer To Peer Options

'on-startup' [function(llc)] This function is called before any attempt to establish peer to peer communication. The *llc* argument provides the `LogicalLinkController` that may be used to allocate and bind listen sockets for local services. The function should return the *llc* object if activation shall continue. Any other value removes the 'llcp' option.

'on-connect' [function(llc)] This function is called when peer to peer communication is successfully established. The *llc* argument provides the now activated `LogicalLinkController` ready for allocation of client communication sockets and data exchange in separate work threads. The function should a true value return more or less immediately, unless it wishes to handle the logical link controller run loop by itself and anytime later return a false value.

'on-release' [function(llc)] This function is called when the symmetry loop was run (the 'on-connect' function returned a true value) and determined that communication with the remote peer has become impossible, or when the 'terminate' function returned a true value. The *llc* object may be used for cleanup actions but not for communication.

'role' [string] This attribute determines whether the local device will restrict itself to either `'initiator'` or `'target'` mode of operation. As Initiator the local device will try to discover a remote device. As Target it waits for being discovered. The default is to alternate between both roles.

'miu' [integer] This attribute sets the maximum information unit size that is announced to the remote device during link activation. The default and also smallest possible value is 128 bytes.

'lto' [integer] This attribute sets the link timeout value (given in milliseconds) that is announced to the remote device during link activation. It informs the remote device that if the local device does not return a protocol data unit before the timeout expires, the communication link is broken and can not be recovered. The *lto* is an important part of the user experience, it ultimately tells when the user should no longer expect communication to continue. The default value is 500 millisecond.

'agf' [boolean] Some early phone implementations did not properly handle aggregated protocol data units. This attribute allows to disable the use af aggregation at the cost of efficiency. Aggregation is disabled with a false value. The default is to use aggregation.

'brs' [integer] For the local device in Initiator role the bit rate selector determines the the bitrate to negotiate with the remote Target. The value may be 0, 1, or 2 for 106, 212, or 424 kbps, respectively. The default is to negotiate 424 kbps.

'acm' [boolean] For the local device in Initiator role this attribute determines whether a remote Target may also be activated in active communication mode. In active communication mode both peer devices mutually generate a radio field when sending. The default is to use passive communication mode.

'rwt' [float] For the local device in Target role this attribute sets the response waiting time announced during link activation. The response waiting time is a medium access layer (NFC-DEP) value that indicates when the remote Initiator shall attempt error recovery after missing a Target response. The value is the waiting time index *wt* that determines the effective response waiting time by the formula `rwt = 4096/13.56E6 * pow(2, wt)`. The value shall not be greater than 14. The default value is 8 and yields an effective response waiting time of 77.33 ms.

'lri' [integer] For the local device in Initiator role this attribute sets the length reduction for medium access layer (NFC-DEP) information frames. The value may be 0, 1, 2, or 3 for a maximum payload size of 64, 128, 192, or 254 bytes, respectively. The default value is 3.

'lrt' [integer] For the local device in Target role this attribute sets the length reduction for medium access layer (NFC-DEP) information frames. The value may be 0, 1, 2, or 3 for a maximum payload size of 64, 128, 192, or 254 bytes, respectively. The default value is 3.

```
import nfc
import nfc.llcp
import threading

def server(socket):
    message, address = socket.recvfrom()
    socket.sendto("It's me!", address)
    socket.close()

def client(socket):
    socket.sendto("Hi there!", address=32)
    socket.close()

def on_startup(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.LOGICAL_DATA_LINK)
    socket.bind(address=32)
    threading.Thread(target=server, args=(socket,)).start()
    return llc

def on_connect(llc):
    socket = nfc.llcp.Socket(llc, nfc.llcp.LOGICAL_DATA_LINK)
    threading.Thread(target=client, args=(socket,)).start()
    return True

llcp_options = {
    'on-startup': on_startup,
    'on-connect': on_connect,
}
with nfc.ContactlessFrontend('usb') as clf:
    clf.connect(llcp=llcp_options)
    print("link terminated")
```

Card Emulation Options

'on-startup' [function(target)] This function is called to prepare a local target for discovery. The input argument is a fresh instance of an unspecific *LocalTarget* that can be set to the desired bitrate and modulation type and populated with the type specific discovery responses (see *listen()* for response data that is needed). The fully specified target object must then be returned.

'on-discover' [function(target)] This function is called when the *LocalTarget* has been discovered. The *target* argument contains the technology type specific discovery commands. The target will be further activated only if this function returns a true value. The default function always returns True.

'on-connect' [function(tag)] This function is called when the local target was discovered and a *nfc.tag.TagEmulation* object successfully initialized. The function receives the emulated *tag* object which stores the first command received after inialization as `tag.cmd`. The function should return a true value if the tag.process_command() and tag.send_response() methods shall be called repeatedly until either the remote device terminates communication or the 'terminate' function returns a true value. The function should return a false value if the *connect()* method shall return immediately with the emulated *tag* object.

'on-release' [function(tag)] This function is called when the Target was released by the Initiator or simply moved away, or if the terminate callback function has returned a true value. The emulated *tag* object may be used for cleanup actions but not for communication.

```
import nfc

def on_startup(target):
    idm = bytearray.fromhex("01010501b00ac30b")
    pmm = bytearray.fromhex("03014b024f4993ff")
    sys = bytearray.fromhex("1234")
    target.brty = "212F"
    target.sensf_res = chr(1) + idm + pmm + sys
    return target

def on_connect(tag):
    print("discovered by remote reader")
    return True

def on_release(tag):
    print("remote reader is gone")
    return True

card_options = {
    'on-startup': on_startup,
    'on-connect': on_connect,
    'on-release': on_release,
}
with nfc.ContactlessFrontend('usb') as clf:
    clf.connect(card=card_options)
```

Return Value

The *connect()* method returns `None` if there were no options left after the 'on-startup' functions have been executed or when the 'terminate' function returned a true value. It returns `False` when terminated by any of the following exceptions: `KeyboardInterrupt`, `IOError`, *UnsupportedTargetError*.

The *connect()* method returns a *Tag*, *LogicalLinkController*, or *TagEmulation* object if the associated 'on-connect' function returned a false value to indicate that it will handle presence check, peer to peer symmetry loop, or command/response processing by itself.

sense(**targets*, ***options*)

Discover a contactless card or listening device.

Note: The *sense()* method is intended for experts with a good understanding of the commands and responses exchanged during target activation (the notion used for commands and responses follows the NFC Forum Digital Specification). If the greater level of control is not needed it is recommended to use the *connect()* method.

All positional arguments build the list of potential *targets* to discover and must be of type *RemoteTarget*. Keyword argument *options* may be the number of `iterations` of the sense loop set by *targets* and the `interval` between iterations. The return value is either a *RemoteTarget* instance or `None`.

```
>>> import nfc, nfc.clf
>>> from binascii import hexlify
>>> clf = nfc.ContactlessFrontend("usb")
>>> target1 = nfc.clf.RemoteTarget("106A")
>>> target2 = nfc.clf.RemoteTarget("212F")
>>> print(clf.sense(target1, target2, iterations=5, interval=0.2))
106A(sdd_res=04497622D93881, sel_res=00, sens_res=4400)
```

A **Type A Target** is specified with the technology letter `A` following the bitrate to be used for the SENS_REQ command (almost always must the bitrate be 106 kbps). To discover only a specific Type A target, the NFCID1 (UID) can be set with a 4, 7, or 10 byte `sel_req` attribute (cascade tags are handled internally).

```
>>> target = nfc.clf.RemoteTarget("106A")
>>> print(clf.sense(target))
106A sdd_res=04497622D93881 sel_res=00 sens_res=4400
>>> target.sel_req = bytearray.fromhex("04497622D93881")
>>> print(clf.sense(target))
106A sdd_res=04497622D93881 sel_res=00 sens_res=4400
>>> target.sel_req = bytearray.fromhex("04497622")
>>> print(clf.sense(target))
None
```

A **Type B Target** is specified with the technology letter `B` following the bitrate to be used for the SENSB_REQ command (almost always must the bitrate be 106 kbps). A specific application family identifier can be set with the first byte of a `sensb_req` attribute (the second byte PARAM is ignored when it can not be set to local device, 00h is a safe value in all cases).

```
>>> target = nfc.clf.RemoteTarget("106B")
>>> print(clf.sense(target))
106B sens_res=50E5DD3DC900000011008185
>>> target.sensb_req = bytearray.fromhex("0000")
>>> print(clf.sense(target))
106B sens_res=50E5DD3DC900000011008185
>>> target.sensb_req = bytearray.fromhex("FF00")
>>> print(clf.sense(target))
None
```

A **Type F Target** is specified with the technology letter `F` following the bitrate to be used for the SENSF_REQ command (the typically supported bitrates are 212 and 424 kbps). The default SENSF_REQ command allows all targets to answer, requests system code information, and selects a single time slot for the SENSF_RES response. This can be changed with the `sensf_req` attribute.

```
>>> target = nfc.clf.RemoteTarget("212F")
>>> print(clf.sense(target))
212F sensf_res=0101010601B00ADE0B03014B024F4993FF12FC
>>> target.sensf_req = bytearray.fromhex("0012FC0000")
>>> print(clf.sense(target))
212F sensf_res=0101010601B00ADE0B03014B024F4993FF
>>> target.sensf_req = bytearray.fromhex("00ABCD0000")
>>> print(clf.sense(target))
None
```

An **Active Communication Mode P2P Target** search is selected with an `atr_req` attribute. The choice of bitrate and modulation type is 106A, 212F, and 424F.

```
>>> atr = bytearray.fromhex("D4000102030405060708091000000030")
>>> target = clf.sense(nfc.clf.RemoteTarget("106A", atr_req=atr))
>>> if target and target.atr_res:
>>>     print(hexlify(target.atr_res).decode())
d501c023cae6b3182afe3dee0000000e3246666d01011103020013040196
>>> target = clf.sense(nfc.clf.RemoteTarget("424F", atr_req=atr))
>>> if target and target.atr_res:
>>>     print(hexlify(target.atr_res).decode())
d501dc0104f04584e15769700000000e3246666d01011103020013040196
```

Some drivers must modify the ATR_REQ to cope with hardware limitations, for example change length reduction value to reduce the maximum size of target responses. The ATR_REQ that has been send is given by the `atr_req` attribute of the returned RemoteTarget object.

A **Passive Communication Mode P2P Target** responds to 106A discovery with bit 6 of SEL_RES set to 1, and to 212F/424F discovery (when the request code RC is 0 in the SENSF_REQ command) with an NFCID2 that starts with 01FEh in the SENSF_RES response. Responses below are from a Nexus 5 configured for NFC-DEP Protocol (SEL_RES bit 6 is set) and Type 4A Tag (SEL_RES bit 5 is set).

```
>>> print(clf.sense(nfc.clf.RemoteTarget("106A")))
106A sdd_res=08796BEB sel_res=60 sens_res=0400
>>> sensf_req = bytearray.fromhex("00FFFF0000")
>>> print(clf.sense(nfc.clf.RemoteTarget("424F", sensf_req=sensf_req)))
424F sensf_res=0101FE1444EFB88FD50000000000000000
```

Errors found in the *targets* argument list raise exceptions only if exactly one target is given. If multiple targets are provided, any target that is not supported or has invalid attributes is just ignored (but is logged as a debug message).

Exceptions

- `IOError` (ENODEV) when a local contacless communication device has not been opened or communication with the local device is no longer possible.
- *`nfc.clf.UnsupportedTargetError`* if the single target supplied as input is not supported by the active driver. This exception is never raised when *`sense()`* is called with multiple targets, those unsupported are then silently ignored.

`listen`(*target*, *timeout*)

Listen *timeout* seconds to become activated as *target*.

Note: The *`listen()`* method is intended for experts with a good understanding of the commands and responses exchanged during target activation (the notion used for commands and responses follows the NFC Forum Digital Specification). If the greater level of control is not needed it is recommended to use the *`connect()`* method.

The *target* argument is a *`LocalTarget`* object that provides bitrate, technology type and response data attributes. The return value is either a *`LocalTarget`* object with bitrate, technology type and request/response data attributes or `None`.

An **P2P Target** is selected when the `atr_res` attribute is set. The bitrate and technology type are decided by the Initiator and do not need to be specified. The `sens_res`, `sdd_res` and `sel_res` attributes for Type A technology as well as the `sensf_res` attribute for Type F technolgy must all be set.

When activated, the bitrate and type are set to the current communication values, the `atr_req` attribute contains the ATR_REQ received from the Initiator and the `dep_req` attribute contains the first DEP_REQ received after activation. If the Initiator has changed communication parameters, the `psl_req` attribute holds the PSL_REQ that was received. The `atr_res` (and the `psl_res` if transmitted) are also made available.

If the local target was activated in passive communication mode either the Type A response (`sens_res`, `sdd_res`, `sel_res`) or Type F response (`sensf_res`) attributes will be present.

With a Nexus 5 on a reader connected via USB the following code should be working and produce similar output (the Nexus 5 prioritizes active communication mode):

```
>>> import nfc, nfc.clf
>>> clf = nfc.ContactlessFrontend("usb")
```

(continues on next page)

(continued from previous page)

```
>>> atr_res = "d50101fe0102030405060708000000083246666d010110"
>>> target = nfc.clf.LocalTarget()
>>> target.sensf_res = bytearray.fromhex("0101FE"+16*"FF")
>>> target.sens_res = bytearray.fromhex("0101")
>>> target.sdd_res = bytearray.fromhex("08010203")
>>> target.sel_res = bytearray.fromhex("40")
>>> target.atr_res = bytearray.fromhex(atr_res)
>>> print(clf.listen(target, timeout=2.5))
424F atr_res=D50101FE0102030405060708000000083246666D010110 ...
```

A **Type A Target** is selected when `atr_res` is not present and the technology type is `A`. The bitrate should be set to 106 kbps, even if a driver supports higher bitrates they would need to be set after activation. The `sens_res`, `sdd_res` and `sel_res` attributes must all be provided.

```
>>> target = nfc.clf.Localtarget("106A")
>>> target.sens_res = bytearray.fromhex("0101"))
>>> target.sdd_res = bytearray.fromhex("08010203")
>>> target.sel_res = bytearray.fromhex("00")
>>> print(clf.listen(target, timeout=2.5))
106A sdd_res=08010203 sel_res=00 sens_res=0101 tt2_cmd=3000
```

A **Type B Target** is selected when `atr_res` is not present and the technology type is `B`. Unfortunately none of the supported devices supports Type B technology for listen and an *nfc.clf.UnsupportedTargetError* exception will be the only result.

```
>>> target = nfc.clf.LocalTarget("106B")
>>> try: clf.listen(target, 2.5)
... except nfc.clf.UnsupportedTargetError: print("sorry")
...
sorry
```

A **Type F Target** is selected when `atr_res` is not present and the technology type is `F`. The bitrate may be 212 or 424 kbps. The `sensf_res` attribute must be provided.

```
>>> idm, pmm, sys = "02FE010203040506", "FFFFFFFFFFFFFFFF", "12FC"
>>> target = nfc.clf.LocalTarget("212F")
>>> target.sensf_res = bytearray.fromhex("01" + idm + pmm + sys)
>>> print(clf.listen(target, 2.5))
212F sensf_req=00FFFF0003 tt3_cmd=0C02FE010203040506 ...
```

Exceptions

- `IOError` (ENODEV) when a local contactless communication device has not been opened or communication with the local device is no longer possible.
- *nfc.clf.UnsupportedTargetError* if the single target supplied as input is not supported by the active driver. This exception is never raised when *sense()* is called with multiple targets, those unsupported are then silently ignored.

exchange (*send_data*, *timeout*)

Exchange data with an activated target (*send_data* is a command frame) or as an activated target (*send_data* is a response frame). Returns a target response frame (if data is send to an activated target) or a next command frame (if data is send from an activated target). Returns None if the communication link broke during exchange (if data is sent as a target). The timeout is the number of seconds to wait for data to return, if the timeout expires an nfc.clf.TimeoutException is raised. Other nfc.clf.CommunicationError exceptions may be raised if an error is detected during communication.

max_send_data_size
The maximum number of octets that can be send with the *exchange()* method in the established operating mode.

max_recv_data_size
The maximum number of octets that can be received with the *exchange()* method in the established operating mode.

7.2.2 Technology Types

class nfc.clf.**RemoteTarget**(*brty*, ***kwargs*)
Bases: nfc.clf.Target

A RemoteTarget instance provides bitrate and technology type and command/response data of a remote card or device that, when input to `sense()`, shall be attempted to discover and, when returned by `sense()`, has been discovered by the local device. Command/response data attributes, whatever name, default to None.

brty
A string that combines bitrate and technology type, e.g. '106A'.

class nfc.clf.**LocalTarget**(*brty='106A'*, ***kwargs*)
Bases: nfc.clf.Target

A LocalTarget instance provides bitrate and technology type and command/response data of the local card or device that, when input to `listen()`, shall be made available for discovery and, when returned by `listen()`, has been discovered by a remote device. Command/response data attributes, whatever name, default to None.

brty
A string that combines bitrate and technology type, e.g. '106A'.

7.2.3 Exceptions

exception nfc.clf.**Error**
Bases: Exception

Base class for exceptions specific to the contacless frontend module.

- UnsupportedTargetError
- CommunicationError
  - ProtocolError
  - TransmissionError
  - TimeoutError
  - BrokenLinkError

exception nfc.clf.**UnsupportedTargetError**
Bases: *nfc.clf.Error*

The *RemoteTarget* input to *ContactlessFrontend.sense()* or *LocalTarget* input to *ContactlessFrontend.listen()* is not supported by the local device.

exception nfc.clf.**CommunicationError**
Bases: *nfc.clf.Error*

Base class for communication errors.

exception nfc.clf.**ProtocolError**
Bases: *nfc.clf.CommunicationError*

Raised when an NFC Forum Digital Specification protocol error occured.

exception nfc.clf.**TransmissionError**
Bases: *nfc.clf.CommunicationError*

Raised when an NFC Forum Digital Specification transmission error occured.

exception nfc.clf.**TimeoutError**
Bases: *nfc.clf.CommunicationError*

Raised when an NFC Forum Digital Specification timeout error occured.

exception nfc.clf.**BrokenLinkError**
Bases: *nfc.clf.CommunicationError*

The remote device (Reader/Writer or P2P Device) has deactivated the RF field or is no longer within communication distance.

7.2.4 Driver Interface

All contactless drivers must implement the interface defined in *Device*. Unsupported target discovery or target emulation methods raise *UnsupportedTargetError*. The interface is used internally by *ContactlessFrontend* and is not intended as an application programming interface. Device driver methods are not thread-safe and do not necessarily check input arguments when they are supposed to be valid. The interface may change without notice at any time.

nfc.clf.device.**connect**(*path*)
Connect to a local device identified by *path* and load the appropriate device driver. The *path* argument is documented at *nfc.clf.ContactlessFrontend.open()*. The return value is either a *Device* instance or None. Note that not all drivers can be autodetected, specifically for serial devices *path* must usually also specify the driver.

class nfc.clf.device.**Device**(**args*, ***kwargs*)
Bases: object

All device drivers inherit from the *Device* class and must implement it's methods.

vendor_name
The device vendor name. An empty string if the vendor name could not be determined.

product_name
The device product name. An empty string if the product name could not be determined.

chipset_name
The name of the chipset embedded in the device.

mute()
Mutes all existing communication, most notably the device will no longer generate a 13.56 MHz carrier signal when operating as Initiator.

sense_tta(*target*)
Discover a Type A Target.

Activates the 13.56 MHz carrier signal and sends a SENS_REQ command at the bitrate set by **target.brty**. If a response is received, sends an RID_CMD for a Type 1 Tag or SDD_REQ and SEL_REQ for a Type 2/4 Tag and returns the responses.

Parameters **target** (nfc.clf.RemoteTarget) – Supplies bitrate and optional command data for the target discovery. The only sensible command to set is **sel_req** populated with a UID to find only that specific target.

Returns

Response data received from a remote target if found. This includes at least **sens_res** and either **rid_res** (for a Type 1 Tag) or **sdd_res** and **sel_res** (for a Type 2/4 Tag).

Return type *nfc.clf.RemoteTarget*

Raises *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).

sense_ttb(*target*)

Discover a Type B Target.

Activates the 13.56 MHz carrier signal and sends a SENSB_REQ command at the bitrate set by **target.brty**. If a SENSB_RES is received, returns a target object with the **sensb_res** attribute.

Note that the firmware of some devices (least all those based on PN53x) automatically sends an ATTRIB command with varying but always unfortunate communication settings. The drivers correct that situation by sending S(DESELECT) and WUPB before return.

Parameters **target** (nfc.clf.RemoteTarget) – Supplies bitrate and the optional **sensb_req** for target discovery. Most drivers do no not allow a fully customized SENSB_REQ, the only parameter that can always be changed is the AFI byte, others may be ignored.

Returns

Response data received from a remote target if found. The only response data attribute is **sensb_res**.

Return type *nfc.clf.RemoteTarget*

Raises *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).

sense_ttf(*target*)

Discover a Type F Target.

Activates the 13.56 MHz carrier signal and sends a SENSF_REQ command at the bitrate set by **target.brty**. If a SENSF_RES is received, returns a target object with the **sensf_res** attribute.

Parameters **target** (nfc.clf.RemoteTarget) – Supplies bitrate and the optional **sensf_req** for target discovery. The default SENSF_REQ invites all targets to respond and requests the system code information bytes.

Returns

Response data received from a remote target if found. The only response data attribute is **sensf_res**.

Return type *nfc.clf.RemoteTarget*

Raises *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).

sense_dep(*target*)

Discover a NFC-DEP Target in active communication mode.

Activates the 13.56 MHz carrier signal and sends an ATR_REQ command at the bitrate set by **target.brty**. If an ATR_RES is received, returns a target object with the **atr_res** attribute.

Note that some drivers (like pn531) may modify the transport data bytes length reduction value in ATR_REQ and ATR_RES due to hardware limitations.

Parameters **target** (nfc.clf.RemoteTarget) – Supplies bitrate and the mandatory **atr_req** for target discovery. The bitrate may be one of '106A', '212F', or '424F'.

Returns

Response data received from a remote target if found. The only response data attribute is **atr_res**. The actually sent and potentially modified ATR_REQ is also included as **atr_req** attribute.

Return type *nfc.clf.RemoteTarget*

Raises *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).

listen_tta(*target*, *timeout*)

Listen as Type A Target.

Waits to receive a SENS_REQ command at the bitrate set by **target.brty** and sends the **target.sens_res** response. Depending on the SENS_RES bytes, the Initiator then sends an RID_CMD (SENS_RES coded for a Type 1 Tag) or SDD_REQ and SEL_REQ (SENS_RES coded for a Type 2/4 Tag). Responses are then generated from the **rid_res** or **sdd_res** and **sel_res** attributes in *target*.

Note that none of the currently supported hardware can actually receive an RID_CMD, thus Type 1 Tag emulation is impossible.

Parameters

- **target** (nfc.clf.LocalTarget) – Supplies bitrate and mandatory response data to reply when being discovered.
- **timeout** (*float*) – The maximum number of seconds to wait for a discovery command.

Returns

Command data received from the remote Initiator if being discovered and to the extent supported by the device. The first command received after discovery is returned as one of the **tt1_cmd**, **tt2_cmd** or **tt4_cmd** attribute (note that unset attributes are always None).

Return type *nfc.clf.LocalTarget*

Raises

- *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).
- ValueError – A required target response attribute is not present or does not supply the number of bytes expected.

listen_ttb(*target*, *timeout*)

Listen as Type A Target.

Waits to receive a SENSB_REQ command at the bitrate set by **target.brty** and sends the **target.sensb_res** response.

Note that none of the currently supported hardware can actually listen as Type B target.

Parameters

- **target** (nfc.clf.LocalTarget) – Supplies bitrate and mandatory response data to reply when being discovered.
- **timeout** (*float*) – The maximum number of seconds to wait for a discovery command.

Returns

Command data received from the remote Initiator if being discovered and to the extent supported by the device. The first command received after discovery is returned as **tt4_cmd** attribute.

Return type *nfc.clf.LocalTarget*

Raises

- *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).
- ValueError – A required target response attribute is not present or does not supply the number of bytes expected.

listen_ttf(*target*, *timeout*)

Listen as Type A Target.

Waits to receive a SENSF_REQ command at the bitrate set by **target.brty** and sends the **target.sensf_res** response. Then waits for a first command that is not a SENSF_REQ and returns this as the **tt3_cmd** attribute.

Parameters

- **target** (nfc.clf.LocalTarget) – Supplies bitrate and mandatory response data to reply when being discovered.
- **timeout** (*float*) – The maximum number of seconds to wait for a discovery command.

Returns

Command data received from the remote Initiator if being discovered and to the extent supported by the device. The first command received after discovery is returned as **tt3_cmd** attribute.

Return type *nfc.clf.LocalTarget*

Raises

- *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).
- ValueError – A required target response attribute is not present or does not supply the number of bytes expected.

listen_dep(*target*, *timeout*)

Listen as NFC-DEP Target.

Waits to receive an ATR_REQ (if the local device supports active communication mode) or a Type A or F Target activation followed by an ATR_REQ in passive communication mode. The ATR_REQ is replied with **target.atr_res**. The first DEP_REQ command is returned as the **dep_req** attribute along with **atr_req** and **atr_res**. The **psl_req** and **psl_res** attributes are returned when the has Initiator performed a parameter selection. The **sens_res** or **sensf_res** attributes are returned when activation was in passive communication mode.

Parameters

- **target** (nfc.clf.LocalTarget) – Supplies mandatory response data to reply when being discovered. All of **sens_res**, **sdd_res**, **sel_res**, **sensf_res**, and **atr_res** must be provided. The bitrate does not need to be set, an NFC-DEP Target always accepts discovery at '106A', '212F and '424F'.
- **timeout** (*float*) – The maximum number of seconds to wait for a discovery command.

Returns

Command data received from the remote Initiator if being discovered and to the extent supported by the device. The first command received after discovery is returned as **dep_req** attribute.

Return type *nfc.clf.LocalTarget*

Raises

- *nfc.clf.UnsupportedTargetError* – The method is not supported by the local hardware.
- ValueError – A required target response attribute is not present or does not supply the number of bytes expected.

send_cmd_recv_rsp(*target*, *data*, *timeout*)
Exchange data with a remote Target

Sends command *data* to the remote *target* discovered in the most recent call to one of the sense_xxx() methods. Note that *target* becomes invalid with any call to mute(), sense_xxx() or listen_xxx()

Parameters

- **target** (nfc.clf.RemoteTarget) – The target returned by the last successful call of a sense_xxx() method.
- **data** (*bytearray*) – The binary data to send to the remote device.
- **timeout** (*float*) – The maximum number of seconds to wait for response data from the remote device.

Returns Response data received from the remote device.

Return type bytearray

Raises *nfc.clf.CommunicationError* – When no data was received.

send_rsp_recv_cmd(*target*, *data*, *timeout=None*)
Exchange data with a remote Initiator

Sends response *data* as the local *target* being discovered in the most recent call to one of the listen_xxx() methods. Note that *target* becomes invalid with any call to mute(), sense_xxx() or listen_xxx()

Parameters

- **target** (nfc.clf.LocalTarget) – The target returned by the last successful call of a listen_xxx() method.
- **data** (*bytearray*) – The binary data to send to the remote device.
- **timeout** (*float*) – The maximum number of seconds to wait for command data from the remote device.

Returns Command data received from the remote device.

Return type bytearray

Raises *nfc.clf.CommunicationError* – When no data was received.

get_max_send_data_size(*target*)
Returns the maximum number of data bytes for sending.

The maximum number of data bytes acceptable for sending with either *send_cmd_recv_rsp()* or *send_rsp_recv_cmd()*. The value reflects the local device capabilities for sending in the mode determined by *target*. It does not relate to any protocol capabilities and negotiations.

Parameters **target** (*nfc.clf.Target*) – The current local or remote communication target.

Returns Maximum number of data bytes supported for sending.

Return type int

get_max_recv_data_size(*target*)
Returns the maximum number of data bytes for receiving.

The maximum number of data bytes acceptable for receiving with either *send_cmd_recv_rsp()* or *send_rsp_recv_cmd()*. The value reflects the local device capabilities for receiving in the mode determined by *target*. It does not relate to any protocol capabilities and negotiations.

Parameters **target** (*nfc.clf.Target*) – The current local or remote communication target.

Returns Maximum number of data bytes supported for receiving.

Return type int

turn_on_led_and_buzzer()
If a device has an LED and/or a buzzer, this method can be implemented to turn those indicators to the ON state.

turn_off_led_and_buzzer()
If a device has an LED and/or a buzzer, this method can be implemented to turn those indicators to the OFF state.

7.2.5 Device Drivers

rcs380

Driver module for contactless devices based on the Sony NFC Port-100 chipset. The only product known to use this chipset is the PaSoRi RC-S380. The RC-S380 connects to the host as a native USB device.

The RC-S380 has been the first NFC Forum certified device. It supports reading and writing of all NFC Forum tags as well as peer-to-peer mode. In addition, the NFC Port-100 also supports card emulation Type A and Type F Technology. A notable restriction is that peer-to-peer active communication mode (not required for NFC Forum certification) is not supported.

function	support	remarks
sense_tta	yes	
sense_ttb	yes	
sense_ttf	yes	
sense_dep	no	
listen_tta	yes	Type F responses can not be disabled
listen_ttb	no	
listen_ttf	yes	
listen_dep	yes	Only passive communication mode

exception nfc.clf.rcs380.**CommunicationError**(*status_bytes*)
Bases: Exception

exception nfc.clf.rcs380.**StatusError**(*status*)
Bases: Exception

class nfc.clf.rcs380.**Device**(*chipset*, *logger*)
Bases: *nfc.clf.device.Device*

mute()
Mutes all existing communication, most notably the device will no longer generate a 13.56 MHz carrier signal when operating as Initiator.

sense_tta(*target*)
Sense for a Type A Target is supported for 106, 212 and 424 kbps. However, there may not be any target that understands the activation commands in other than 106 kbps.

sense_ttb(*target*)
Sense for a Type B Target is supported for 106, 212 and 424 kbps. However, there may not be any target that understands the activation command in other than 106 kbps.

sense_ttf(*target*)
Sense for a Type F Target is supported for 212 and 424 kbps.

sense_dep(*target*)
Sense for an active DEP Target is not supported. The device only supports passive activation via sense_tta/sense_ttf.

listen_tta(*target*, *timeout*)
Listen as Type A Target in 106 kbps.

Restrictions:

- It is not possible to send short frames that are required for ACK and NAK responses. This means that a Type 2 Tag emulation can only implement a single sector memory model.
- It can not be avoided that the chipset responds to SENSF_REQ commands. The driver configures the SENSF_RES response to all zero and ignores all Type F communication but eventually it depends on the remote device whether Type A Target activation will still be attempted.

listen_ttb(*target*, *timeout*)
Listen as Type B Target is not supported.

listen_ttf(*target*, *timeout*)
Listen as Type F Target is supported for either 212 or 424 kbps.

listen_dep(*target*, *timeout*)
Listen as NFC-DEP Target.

Waits to receive an ATR_REQ (if the local device supports active communication mode) or a Type A or F Target activation followed by an ATR_REQ in passive communication mode. The ATR_REQ is replied with **target.atr_res**. The first DEP_REQ command is returned as the **dep_req** attribute along with **atr_req** and **atr_res**. The **psl_req** and **psl_res** attributes are returned when the has Initiator performed a parameter selection. The **sens_res** or **sensf_res** attributes are returned when activation was in passive communication mode.

Parameters

- **target** (*nfc.clf.LocalTarget*) – Supplies mandatory response data to reply when being discovered. All of **sens_res**, **sdd_res**, **sel_res**, **sensf_res**, and **atr_res** must be provided. The bitrate does not need to be set, an NFC-DEP Target always accepts discovery at '106A', '212F and '424F'.
- **timeout** (*float*) – The maximum number of seconds to wait for a discovery command.

Returns

Command data received from the remote Initiator if being discovered and to the extent supported by the device. The first command received after discovery is returned as **dep_req** attribute.

Return type *nfc.clf.LocalTarget*

Raises

- *nfc.clf.UnsupportedTargetError* – The method is not supported by the local hardware.
- ValueError – A required target response attribute is not present or does not supply the number of bytes expected.

get_max_send_data_size(*target*)

Returns the maximum number of data bytes for sending.

The maximum number of data bytes acceptable for sending with either *send_cmd_recv_rsp()* or *send_rsp_recv_cmd()*. The value reflects the local device capabilities for sending in the mode determined by *target*. It does not relate to any protocol capabilities and negotiations.

Parameters **target** (*nfc.clf.Target*) – The current local or remote communication target.

Returns Maximum number of data bytes supported for sending.

Return type int

get_max_recv_data_size(*target*)

Returns the maximum number of data bytes for receiving.

The maximum number of data bytes acceptable for receiving with either *send_cmd_recv_rsp()* or *send_rsp_recv_cmd()*. The value reflects the local device capabilities for receiving in the mode determined by *target*. It does not relate to any protocol capabilities and negotiations.

Parameters **target** (*nfc.clf.Target*) – The current local or remote communication target.

Returns Maximum number of data bytes supported for receiving.

Return type int

send_cmd_recv_rsp(*target*, *data*, *timeout*)

Exchange data with a remote Target

Sends command *data* to the remote *target* discovered in the most recent call to one of the sense_xxx() methods. Note that *target* becomes invalid with any call to mute(), sense_xxx() or listen_xxx()

Parameters

- **target** (nfc.clf.RemoteTarget) – The target returned by the last successful call of a sense_xxx() method.
- **data** (*bytearray*) – The binary data to send to the remote device.
- **timeout** (*float*) – The maximum number of seconds to wait for response data from the remote device.

Returns Response data received from the remote device.

Return type bytearray

Raises *nfc.clf.CommunicationError* – When no data was received.

send_rsp_recv_cmd(*target*, *data*, *timeout*)
Exchange data with a remote Initiator

Sends response *data* as the local *target* being discovered in the most recent call to one of the listen_xxx() methods. Note that *target* becomes invalid with any call to mute(), sense_xxx() or listen_xxx()

Parameters

- **target** (nfc.clf.LocalTarget) – The target returned by the last successful call of a listen_xxx() method.
- **data** (*bytearray*) – The binary data to send to the remote device.
- **timeout** (*float*) – The maximum number of seconds to wait for command data from the remote device.

Returns Command data received from the remote device.

Return type bytearray

Raises *nfc.clf.CommunicationError* – When no data was received.

pn531

Driver module for contactless devices based on the NXP PN531 chipset. This was once a (sort of) joint development between Philips and Sony to supply hardware capable of running the ISO/IEC 18092 Data Exchange Protocol. The chip has selectable UART, I2C, SPI, or USB host interfaces, For USB the vendor and product ID can be switched by a hardware pin to either Philips or Sony.

The internal chipset architecture comprises a small 8-bit MCU and a Contactless Interface Unit CIU that is basically a PN511. The CIU implements the analog and digital part of communication (modulation and framing) while the MCU handles the protocol parts and host communication. The PN511 and hence the PN531 does not support Type B Technology and can not handle the specific Jewel/Topaz (Type 1 Tag) communication. Compared to PN532/PN533 the host frame structure does not allow maximum size ISO/IEC 18092 packets to be transferred. The driver handles this restriction by modifying the initialization commands (ATR, PSL) when needed.

function	support	remarks
sense_tta	yes	Type 1 Tag is not supported
sense_ttb	no	
sense_ttf	yes	
sense_dep	yes	Reduced transport data byte length (max 192)
listen_tta	yes	
listen_ttb	no	
listen_ttf	yes	Maximimum frame size is 64 byte
listen_dep	yes	

class nfc.clf.pn531.**Device**(*chipset*, *logger*)
Bases: nfc.clf.pn53x.Device

sense_tta(*target*)
Activate the RF field and probe for a Type A Target.

The PN531 can discover some Type A Targets (Type 2 Tag and Type 4A Tag) at 106 kbps. Type 1 Tags (Jewel/Topaz) are completely unsupported. Because the firmware does not evaluate the SENS_RES before sending SDD_REQ, it may be that a warning message about missing Type 1 Tag support is logged even if a Type 2 or 4A Tag was present. This typically happens when the SDD_RES or SEL_RES are lost due to communication errors (normally when the tag is moved away).

sense_ttb(*target*)
Sense for a Type B Target is not supported.

sense_ttf(*target*)
Activate the RF field and probe for a Type F Target.

sense_dep(*target*)
Search for a DEP Target in active communication mode.

Because the PN531 does not implement the extended frame syntax for host controller communication, it can not support the maximum payload size of 254 byte. The driver handles this by modifying the length-reduction values in atr_req and atr_res.

listen_tta(*target*, *timeout*)
Listen *timeout* seconds for a Type A activation at 106 kbps. The `sens_res`, `sdd_res`, and `sel_res` response data must be provided and `sdd_res` must be a 4 byte UID that starts with `08h`. Depending on `sel_res` an activation may return a target with a `tt2_cmd`, `tt4_cmd` or `atr_req` attribute. The default RATS response sent for a Type 4 Tag activation can be replaced with a `rats_res` attribute.

listen_ttb(*target*, *timeout*)
Listen as Type B Target is not supported.

listen_ttf(*target*, *timeout*)
Listen *timeout* seconds for a Type F card activation. The target `brty` must be set to either 212F or 424F and `sensf_res` provide 19 byte response data (response code + 8 byte IDm + 8 byte PMm + 2 byte system code). Note that the maximum command an response frame length is 64 bytes only (including the frame length byte), because the driver must directly program the contactless interface unit within the PN533.

listen_dep(*target*, *timeout*)
Listen *timeout* seconds to become initialized as a DEP Target.

The PN531 can be set to listen as a DEP Target for passive and active communication mode.

pn532

Driver module for contactless devices based on the NXP PN532 chipset. This successor of the PN531 can additionally handle Type B Technology (type 4B Tags) and Type 1 Tag communication. It also supports an extended frame syntax for host communication that allows larger packets to be transferred. The chip has selectable UART, I2C or SPI host interfaces. A speciality of the PN532 is that it can manage two targets (cards) simultanously, although this is not used by *nfcpy*.

The internal chipset architecture comprises a small 8-bit MCU and a Contactless Interface Unit CIU that is basically a PN512. The CIU implements the analog and digital part of communication (modulation and framing) while the MCU handles the protocol parts and host communication. Almost all PN532 firmware limitations (or bugs) can be avoided by directly programming the CIU. Type F Target mode for card emulation is completely implemented with the CIU and limited to 64 byte frame exchanges by the CIU's FIFO size. Type B Target mode is not possible.

function	support	remarks
sense_tta	yes	
sense_ttb	yes	
sense_ttf	yes	
sense_dep	yes	
listen_tta	yes	
listen_ttb	no	
listen_ttf	yes	Maximimum frame size is 64 byte
listen_dep	yes	

class nfc.clf.pn532.**Device**(*chipset*, *logger*)
Bases: nfc.clf.pn53x.Device

sense_tta(*target*)
Search for a Type A Target.

The PN532 can discover all kinds of Type A Targets (Type 1 Tag, Type 2 Tag, and Type 4A Tag) at 106 kbps.

sense_ttb(*target*)
Search for a Type B Target.

The PN532 can discover Type B Targets (Type 4B Tag) at 106 kbps. For a Type 4B Tag the firmware automatically sends an ATTRIB command that configures the use of DID and 64 byte maximum frame size. The driver reverts this configuration with a DESELECT and WUPB command to return the target prepared for activation (which nfcpy does in the tag activation code).

sense_ttf(*target*)
Search for a Type F Target.

The PN532 can discover Type F Targets (Type 3 Tag) at 212 and 424 kbps. The driver uses the default polling command `06FFFF0000` if no `target.sens_req` is supplied.

sense_dep(*target*)
Search for a DEP Target in active communication mode.

listen_tta(*target*, *timeout*)
Listen *timeout* seconds for a Type A activation at 106 kbps. The `sens_res`, `sdd_res`, and `sel_res` response data must be provided and `sdd_res` must be a 4 byte UID that starts with `08h`. Depending on `sel_res` an activation may return a target with a `tt2_cmd`, `tt4_cmd` or `atr_req` attribute. The default RATS response sent for a Type 4 Tag activation can be replaced with a `rats_res` attribute.

listen_ttb(*target*, *timeout*)
Listen as Type B Target is not supported.

listen_ttf(*target*, *timeout*)
Listen *timeout* seconds for a Type F card activation. The target `brty` must be set to either 212F or 424F and `sensf_res` provide 19 byte response data (response code + 8 byte IDm + 8 byte PMm + 2 byte system code). Note that the maximum command an response frame length is 64 bytes only (including the frame length byte), because the driver must directly program the contactless interface unit within the PN533.

listen_dep(*target*, *timeout*)
Listen *timeout* seconds to become initialized as a DEP Target.

The PN532 can be set to listen as a DEP Target for passive and active communication mode.

pn533

Driver module for contactless devices based on the NXP PN533 chipset. The PN533 is pretty similar to the PN532 except that it also has a USB host interface option and, probably due to the resources needed for USB, does not support two simultaneous targets. Anything else said about PN532 also applies to PN533.

function	support	remarks
sense_tta	yes	
sense_ttb	yes	
sense_ttf	yes	
sense_dep	yes	
listen_tta	yes	
listen_ttb	no	
listen_ttf	yes	Maximimum frame size is 64 byte
listen_dep	yes	

class nfc.clf.pn533.**Device**(*chipset*, *logger*)
Bases: nfc.clf.pn53x.Device

sense_tta(*target*)
Activate the RF field and probe for a Type A Target.

The PN533 can discover all kinds of Type A Targets (Type 1 Tag, Type 2 Tag, and Type 4A Tag) at 106 kbps.

sense_ttb(*target*)
Activate the RF field and probe for a Type B Target.

The PN533 can discover Type B Targets (Type 4B Tag) at 106, 212, 424, and 848 kbps. The PN533 automatically sends an ATTRIB command that configures a 64 byte maximum frame size. The driver reverts this configuration with a DESELECT and WUPB command to return the target prepared for activation.

sense_ttf(*target*)
Activate the RF field and probe for a Type F Target.

The PN533 can discover Type F Targets (Type 3 Tag) at 212 and 424 kbps.

sense_dep(*target*)
Search for a DEP Target in active communication mode.

send_cmd_recv_rsp(*target*, *data*, *timeout*)
Send command *data* to the remote *target* and return the response data if received within *timeout* seconds.

listen_tta(*target*, *timeout*)
Listen *timeout* seconds for a Type A activation at 106 kbps. The `sens_res`, `sdd_res`, and `sel_res` response data must be provided and `sdd_res` must be a 4 byte UID that starts with `08h`. Depending on `sel_res` an activation may return a target with a `tt2_cmd`, `tt4_cmd` or `atr_req` attribute. The default RATS response sent for a Type 4 Tag activation can be replaced with a `rats_res` attribute.

listen_ttb(*target*, *timeout*)
Listen as Type B Target is not supported.

listen_ttf(*target*, *timeout*)
Listen *timeout* seconds for a Type F card activation. The target `brty` must be set to either 212F or 424F and `sensf_res` provide 19 byte response data (response code + 8 byte IDm + 8 byte PMm + 2 byte system code). Note that the maximum command an response frame length is 64 bytes only (including the frame length byte), because the driver must directly program the contactless interface unit within the PN533.

listen_dep(*target*, *timeout*)
Listen *timeout* seconds to become initialized as a DEP Target.

The PN533 can be set to listen as a DEP Target for passive and active communication mode.

send_rsp_recv_cmd(*target*, *data*, *timeout*)
While operating as *target* send response *data* to the remote device and return new command data if received within *timeout* seconds.

rcs956

Driver for contacless devices based on the Sony RC-S956 chipset. Products known to use this chipset are the PaSoRi RC-S330, RC-S360, and RC-S370. The RC-S956 connects to the host as a native USB device.

The RC-S956 has the same hardware architecture as the NXP PN53x family, i.e. it has a PN512 Contactless Interface Unit (CIU) coupled with a 80C51 microcontroller and uses the same frame structure for host communication and mostly the same commands. However, the firmware that runs on the 80C51 is different and the most notable difference is a much stricter state machine. The state machine restricts allowed commands to certain modes. While direct access to the CIU registers is possible, some of the things that can be done with a PN53x are unfortunately prevented by the stricter state machine.

function	support	remarks
sense_tta	yes	Only Type 1 Tags up to 128 byte (Topaz-96)
sense_ttb	yes	ATTRIB by firmware voided with S(DESELECT)
sense_ttf	yes	
sense_dep	yes	
listen_tta	yes	Only DEP and Type 2 Target
listen_ttb	no	
listen_ttf	no	
listen_dep	yes	Only passive communication mode

class nfc.clf.rcs956.**Device**(*chipset*, *logger*)
Bases: nfc.clf.pn53x.Device

mute()
Mutes all existing communication, most notably the device will no longer generate a 13.56 MHz carrier signal when operating as Initiator.

sense_tta(*target*)
Activate the RF field and probe for a Type A Target.

The RC-S956 can discover all Type A Targets (Type 1 Tag, Type 2 Tag, and Type 4A Tag) at 106 kbps. Due to firmware restrictions it is not possible to read a Type 1 Tag with dynamic memory layout (more than 128 byte memory).

sense_ttb(*target*)
Activate the RF field and probe for a Type B Target.

The RC-S956 can discover Type B Targets (Type 4B Tag) at 106 kbps. For a Type 4B Tag the firmware automatically sends an ATTRIB command that configures the use of DID and 64 byte maximum frame size. The driver reverts this configuration with a DESELECT and WUPB command to return the target prepared for activation (which nfcpy does in the tag activation code).

sense_ttf(*target*)
Activate the RF field and probe for a Type F Target.

sense_dep(*target*)
Search for a DEP Target in active or passive communication mode.

listen_tta(*target*, *timeout*)
Listen *timeout* seconds for a Type A activation at 106 kbps. The sens_res, sdd_res, and sel_res response data must be provided and sdd_res must be a 4 byte UID that starts with 08h. Depending

on `sel_res` an activation may return a target with `tt2_cmd` or `atr_req` attribute. A Type 4A Tag activation is not supported.

listen_ttb(*target*, *timeout*)
Listen as Type B Target is not supported.

listen_ttf(*target*, *timeout*)
Listen as Type F Target is not supported.

listen_dep(*target*, *timeout*)
Listen *timeout* seconds to become initialized as a DEP Target.

The RC-S956 can be set to listen as a DEP Target for passive communication mode. Target active communication mode is disabled by the driver due to performance issues. It is also not possible to fully control the ATR_RES response, only the response waiting time (TO byte of ATR_RES) and the general bytes can be set by the driver. Because the TO value must be set before calling the hardware listen function, it can not be different for the Type A of Type F passive initalization (the driver uses the higher value if they are different).

acr122

Device driver for the Arygon ACR122U contactless reader.

The Arygon ACR122U is a PC/SC compliant contactless reader that connects via USB and uses the USB CCID profile. It is normally intented to be used with a PC/SC stack but this driver interfaces directly with the inbuilt PN532 chipset by tunneling commands through the PC/SC Escape command. The driver is limited in functionality because the embedded microprocessor (that implements the PC/SC stack) also operates the PN532; it does not allow all commands to pass as desired and reacts on chip responses with its own (legitimate) interpretation of state.

function	support	remarks
sense_tta	yes	Type 1 (Topaz) Tags are not supported
sense_ttb	yes	ATTRIB by firmware voided with S(DESELECT)
sense_ttf	yes	
sense_dep	yes	
listen_tta	no	
listen_ttb	no	
listen_ttf	no	
listen_dep	no	

class nfc.clf.acr122.**Device**(*chipset*)
Bases: *nfc.clf.pn532.Device*

sense_tta(*target*)
Activate the RF field and probe for a Type A Target at 106 kbps. Other bitrates are not supported. Type 1 Tags are not supported because the device does not allow to send the correct RID command (even though the PN532 does).

sense_ttb(*target*)
Activate the RF field and probe for a Type B Target.

The RC-S956 can discover Type B Targets (Type 4B Tag) at 106 kbps. For a Type 4B Tag the firmware automatically sends an ATTRIB command that configures the use of DID and 64 byte maximum frame size. The driver reverts this configuration with a DESELECT and WUPB command to return the target prepared for activation (which nfcpy does in the tag activation code).

sense_ttf(*target*)
Activate the RF field and probe for a Type F Target. Bitrates 212 and 424 kpbs are supported.

sense_dep(*target*)
Search for a DEP Target. Both passive and passive communication mode are supported.

listen_tta(*target*, *timeout*)
Listen as Type A Target is not supported.

listen_ttb(*target*, *timeout*)
Listen as Type B Target is not supported.

listen_ttf(*target*, *timeout*)
Listen as Type F Target is not supported.

listen_dep(*target*, *timeout*)
Listen as DEP Target is not supported.

turn_on_led_and_buzzer()
Buzz and turn red.

turn_off_led_and_buzzer()
Back to green.

udp

Driver module for simulated contactless communication over UDP/IP. It can be activated with the device path `udp:<host>:<port>` where the optional *host* may be the IP address or name of the node where the targeted communication partner is listening on *port*. The default values for *host* and *port* are `localhost:54321`.

The driver implements almost all communication modes, with the current exception of active communication mode data exchange protocol.

function	support	remarks
sense_tta	yes	
sense_ttb	yes	
sense_ttf	yes	
sense_dep	no	
listen_tta	yes	
listen_ttb	yes	
listen_ttf	yes	
listen_dep	yes	

class nfc.clf.udp.**Device**(*host*, *port*)
Bases: *nfc.clf.device.Device*

mute()
Mutes all existing communication, most notably the device will no longer generate a 13.56 MHz carrier signal when operating as Initiator.

sense_tta(*target*)
Discover a Type A Target.

Activates the 13.56 MHz carrier signal and sends a SENS_REQ command at the bitrate set by **target.brty**. If a response is received, sends an RID_CMD for a Type 1 Tag or SDD_REQ and SEL_REQ for a Type 2/4 Tag and returns the responses.

Parameters **target** (nfc.clf.RemoteTarget) – Supplies bitrate and optional command data for the target discovery. The only sensible command to set is **sel_req** populated with a UID to find only that specific target.

Returns

Response data received from a remote target if found. This includes at least **sens_res** and either **rid_res** (for a Type 1 Tag) or **sdd_res** and **sel_res** (for a Type 2/4 Tag).

Return type *nfc.clf.RemoteTarget*

Raises `nfc.clf.UnsupportedTargetError` – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).

sense_ttb(*target*)

Discover a Type B Target.

Activates the 13.56 MHz carrier signal and sends a SENSB_REQ command at the bitrate set by **target.brty**. If a SENSB_RES is received, returns a target object with the **sensb_res** attribute.

Note that the firmware of some devices (least all those based on PN53x) automatically sends an ATTRIB command with varying but always unfortunate communication settings. The drivers correct that situation by sending S(DESELECT) and WUPB before return.

Parameters **target** (`nfc.clf.RemoteTarget`) – Supplies bitrate and the optional **sensb_req** for target discovery. Most drivers do no not allow a fully customized SENSB_REQ, the only parameter that can always be changed is the AFI byte, others may be ignored.

Returns

Response data received from a remote target if found. The only response data attribute is **sensb_res**.

Return type *nfc.clf.RemoteTarget*

Raises `nfc.clf.UnsupportedTargetError` – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).

sense_ttf(*target*)

Discover a Type F Target.

Activates the 13.56 MHz carrier signal and sends a SENSF_REQ command at the bitrate set by **target.brty**. If a SENSF_RES is received, returns a target object with the **sensf_res** attribute.

Parameters **target** (`nfc.clf.RemoteTarget`) – Supplies bitrate and the optional **sensf_req** for target discovery. The default SENSF_REQ invites all targets to respond and requests the system code information bytes.

Returns

Response data received from a remote target if found. The only response data attribute is **sensf_res**.

Return type *nfc.clf.RemoteTarget*

Raises `nfc.clf.UnsupportedTargetError` – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).

sense_dep(*target*)

Discover a NFC-DEP Target in active communication mode.

Activates the 13.56 MHz carrier signal and sends an ATR_REQ command at the bitrate set by **target.brty**. If an ATR_RES is received, returns a target object with the **atr_res** attribute.

Note that some drivers (like pn531) may modify the transport data bytes length reduction value in ATR_REQ and ATR_RES due to hardware limitations.

Parameters **target** (nfc.clf.RemoteTarget) – Supplies bitrate and the mandatory **atr_req** for target discovery. The bitrate may be one of '106A', '212F', or '424F'.

Returns

Response data received from a remote target if found. The only response data attribute is **atr_res**. The actually sent and potentially modified ATR_REQ is also included as **atr_req** attribute.

Return type *nfc.clf.RemoteTarget*

Raises *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).

listen_tta(*target*, *timeout*)

Listen as Type A Target.

Waits to receive a SENS_REQ command at the bitrate set by **target.brty** and sends the **target.sens_res** response. Depending on the SENS_RES bytes, the Initiator then sends an RID_CMD (SENS_RES coded for a Type 1 Tag) or SDD_REQ and SEL_REQ (SENS_RES coded for a Type 2/4 Tag). Responses are then generated from the **rid_res** or **sdd_res** and **sel_res** attributes in *target*.

Note that none of the currently supported hardware can actually receive an RID_CMD, thus Type 1 Tag emulation is impossible.

Parameters

- **target** (nfc.clf.LocalTarget) – Supplies bitrate and mandatory response data to reply when being discovered.
- **timeout** (*float*) – The maximum number of seconds to wait for a discovery command.

Returns

Command data received from the remote Initiator if being discovered and to the extent supported by the device. The first command received after discovery is returned as one of the **tt1_cmd**, **tt2_cmd** or **tt4_cmd** attribute (note that unset attributes are always None).

Return type *nfc.clf.LocalTarget*

Raises

- *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).
- ValueError – A required target response attribute is not present or does not supply the number of bytes expected.

listen_ttb(*target*, *timeout*)

Listen as Type A Target.

Waits to receive a SENSB_REQ command at the bitrate set by **target.brty** and sends the **target.sensb_res** response.

Note that none of the currently supported hardware can actually listen as Type B target.

Parameters

- **target** (nfc.clf.LocalTarget) – Supplies bitrate and mandatory response data to reply when being discovered.
- **timeout** (*float*) – The maximum number of seconds to wait for a discovery command.

Returns

Command data received from the remote Initiator if being discovered and to the extent supported by the device. The first command received after discovery is returned as **tt4_cmd** attribute.

Return type *nfc.clf.LocalTarget*

Raises

- *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).
- ValueError – A required target response attribute is not present or does not supply the number of bytes expected.

listen_ttf(*target*, *timeout*)

Listen as Type A Target.

Waits to receive a SENSF_REQ command at the bitrate set by **target.brty** and sends the **target.sensf_res** response. Then waits for a first command that is not a SENSF_REQ and returns this as the **tt3_cmd** attribute.

Parameters

- **target** (nfc.clf.LocalTarget) – Supplies bitrate and mandatory response data to reply when being discovered.
- **timeout** (*float*) – The maximum number of seconds to wait for a discovery command.

Returns

Command data received from the remote Initiator if being discovered and to the extent supported by the device. The first command received after discovery is returned as **tt3_cmd** attribute.

Return type *nfc.clf.LocalTarget*

Raises

- *nfc.clf.UnsupportedTargetError* – The method is not supported or the *target* argument requested an unsupported bitrate (or has a wrong technology type identifier).
- ValueError – A required target response attribute is not present or does not supply the number of bytes expected.

listen_dep(*target*, *timeout*)

Listen as NFC-DEP Target.

Waits to receive an ATR_REQ (if the local device supports active communication mode) or a Type A or F Target activation followed by an ATR_REQ in passive communication mode. The ATR_REQ is replied with **target.atr_res**. The first DEP_REQ command is returned as the **dep_req** attribute along with **atr_req** and **atr_res**. The **psl_req** and **psl_res** attributes are returned when the has Initiator performed a parameter selection. The **sens_res** or **sensf_res** attributes are returned when activation was in passive communication mode.

Parameters

- **target** (nfc.clf.LocalTarget) – Supplies mandatory response data to reply when being discovered. All of **sens_res**, **sdd_res**, **sel_res**, **sensf_res**, and **atr_res** must be provided. The bitrate does not need to be set, an NFC-DEP Target always accepts discovery at '106A', '212F and '424F'.
- **timeout** (*float*) – The maximum number of seconds to wait for a discovery command.

Returns

Command data received from the remote Initiator if being discovered and to the extent supported by the device. The first command received after discovery is returned as **dep_req** attribute.

Return type *nfc.clf.LocalTarget*

Raises

- *nfc.clf.UnsupportedTargetError* – The method is not supported by the local hardware.
- ValueError – A required target response attribute is not present or does not supply the number of bytes expected.

send_cmd_recv_rsp(*target*, *data*, *timeout*)
Exchange data with a remote Target

Sends command *data* to the remote *target* discovered in the most recent call to one of the sense_xxx() methods. Note that *target* becomes invalid with any call to mute(), sense_xxx() or listen_xxx()

Parameters

- **target** (nfc.clf.RemoteTarget) – The target returned by the last successful call of a sense_xxx() method.
- **data** (*bytearray*) – The binary data to send to the remote device.
- **timeout** (*float*) – The maximum number of seconds to wait for response data from the remote device.

Returns Response data received from the remote device.

Return type bytearray

Raises *nfc.clf.CommunicationError* – When no data was received.

send_rsp_recv_cmd(*target*, *data*, *timeout*)
Exchange data with a remote Initiator

Sends response *data* as the local *target* being discovered in the most recent call to one of the listen_xxx() methods. Note that *target* becomes invalid with any call to mute(), sense_xxx() or listen_xxx()

Parameters

- **target** (nfc.clf.LocalTarget) – The target returned by the last successful call of a listen_xxx() method.
- **data** (*bytearray*) – The binary data to send to the remote device.
- **timeout** (*float*) – The maximum number of seconds to wait for command data from the remote device.

Returns Command data received from the remote device.

Return type bytearray

Raises *nfc.clf.CommunicationError* – When no data was received.

get_max_send_data_size(*target*)
Returns the maximum number of data bytes for sending.

The maximum number of data bytes acceptable for sending with either *send_cmd_recv_rsp()* or *send_rsp_recv_cmd()*. The value reflects the local device capabilities for sending in the mode determined by *target*. It does not relate to any protocol capabilities and negotiations.

Parameters **target** (*nfc.clf.Target*) – The current local or remote communication target.

Returns Maximum number of data bytes supported for sending.

Return type int

get_max_recv_data_size(*target*)

Returns the maximum number of data bytes for receiving.

The maximum number of data bytes acceptable for receiving with either *send_cmd_recv_rsp()* or *send_rsp_recv_cmd()*. The value reflects the local device capabilities for receiving in the mode determined by *target*. It does not relate to any protocol capabilities and negotiations.

Parameters **target** (*nfc.clf.Target*) – The current local or remote communication target.

Returns Maximum number of data bytes supported for receiving.

Return type int

7.3 nfc.tag

class nfc.tag.**Tag**(*clf*, *target*)

Bases: object

The base class for all NFC Tags/Cards. The methods and attributes defined here are commonly available but some may, depending on the tag product, also return a None value is support is not available.

Direct subclasses are the NFC Forum tag types: *Type1Tag*, *Type2Tag*, *Type3Tag*, *Type4Tag*. Some of them are further specialized in vendor/product specific classes.

class NDEF(*tag*)

Bases: object

The NDEF object type that may be read from *Tag.ndef*.

This class presents the NDEF management information and the actual NDEF message by a couple of attributes. It is normally accessed from a *Tag* instance (further named *tag*) through the *Tag.ndef* attribute for reading or writing NDEF records.

```
if tag.ndef is not None:
    for record in tag.ndef.records:
        print(record)
    if tag.ndef.is_writeable:
        from ndef import TextRecord
        tag.ndef.records = [TextRecord("Hello World")]
```

tag

A readonly reference to the underlying tag object.

length
Length of the current NDEF message in bytes.

capacity
Maximum number of bytes for an NDEF message.

is_readable
`True` if the NDEF data are is readable.

is_writeable
`True` if the NDEF data area is writeable.

has_changed
The boolean attribute *has_changed* allows to determine whether the NDEF message on the tag is different from the message that was read or written at an earlier time in the session. This may for example be the case if the tag is build to dynamically present different content depending on some state.

Note that reading this attribute involves a complete update of the *Tag.NDEF* instance and it is possible that *Tag.ndef* is `None` after the update (e.g. tag gone during read or a dynamic tag that failed). A robust implementation should always verify the value of the *Tag.ndef* attribute.

```
if tag.ndef.has_changed and tag.ndef is not None:
    for record in tag.ndef.records:
        print(record)
```

The *has_changed* attribute can also be used to verify that NDEF records written to the tag are identical to the NDEF records stored on the tag.

```
from ndef import TextRecord
tag.ndef.records = [TextRecord("Hello World")]
if tag.ndef.has_changed:
    print("the tag data differs from what was written")
```

records
Read or write a list of NDEF Records.

New in version 0.12.

This attribute is a convinience wrapper for decoding and encoding of the NDEF message data *octets*. It uses the ndeflib module to return the list of `ndef.Record` instances decoded from the NDEF message data or set the message data from a list of records.

```
from ndef import TextRecord
if tag.ndef is not None:
    for record in tag.ndef.records:
        print(record)
    try:
        tag.ndef.records = [TextRecord('Hello World')]
    except nfc.tag.TagCommandError as err:
        print("NDEF write failed: " + str(err))
```

Decoding is performed with a relaxed error handling strategy that ignores minor errors in the NDEF data. The ndeflib does also support 'strict' and 'ignore' error handling which may be used like so:

```
from ndef import message_decoder, message_encoder
records = message_decoder(tag.ndef.octets, errors='strict')
tag.ndef.octets = b''.join(message_encoder(records))
```

octets
Read or write NDEF message data octets.

New in version 0.12.

The *octets* attribute returns the NDEF message data octets as bytes. A bytes or bytearray sequence assigned to *octets* is immediately written to the NDEF message data area, unless the Tag memory is write protected or to small.

```
if tag.ndef is not None:
    print(hexlify(tag.ndef.octets).decode())
```

identifier
The unique tag identifier.

ndef
An *NDEF* object if found, otherwise `None`.

is_present
True if the tag is within communication range.

is_authenticated
True if the tag was successfully authenticated.

dump()
The dump() method returns a list of strings describing the memory structure of the tag, suitable for printing with join(). The list format makes custom indentation a bit easier.

```
print("\n".join(["\t" + line for line in tag.dump()]))
```

format(*version=None*, *wipe=None*)
Format the tag to make it NDEF compatible or erase content.

The *format()* method is highly dependent on the tag type, product and present status, for example a tag that has been made read-only with lock bits can no longer be formatted or erased.

format() creates the management information defined by the NFC Forum to describes the NDEF data area on the tag, this is also called NDEF mapping. The mapping may differ between versions of the tag specifications, the mapping to apply can be specified with the *version* argument as an 8-bit integer composed of a major version number in the most significant 4 bit and the minor version number in the least significant 4 bit. If *version* is not specified then the highest possible mapping version is used.

If formatting of the tag is possible, the default behavior of *format()* is to update only the management information required to make the tag appear as NDEF compatible and empty, previously existing data could still be read. If existing data shall be overwritten, the *wipe* argument can be set to an 8-bit integer that will be written to all available bytes.

The *format()* method returns `True` if formatting was successful, `False` if it failed for some reason, or `None` if the present tag can not be formatted either because the tag does not support formatting or it is not implemented in nfcpy.

protect(*password=None*, *read_protect=False*, *protect_from=0*)
Protect a tag against future write or read access.

protect() attempts to make a tag readonly for all readers if *password* is `None`, writeable only after authentication if a *password* is provided, and readable only after authentication if a *password* is provided and the *read_protect* flag is set. The *password* must be a byte or character sequence that provides sufficient key material for the tag specific protect function (this is documented separately for the individual tag types). As a special case, if *password* is set to an empty string the *protect()* method uses a default manufacturer value if such is known.

The *protect_from* argument sets the first memory unit to be protected. Memory units are tag type specific, for a Type 1 or Type 2 Tag a memory unit is 4 byte, for a Type 3 Tag it is 16 byte, and for a Type 4 Tag it is the complete NDEF data area.

Note that the effect of protecting a tag without password can normally not be reversed.

The return value of *protect()* is either `True` or `False` depending on whether the operation was successful or not, or `None` if the tag does not support custom protection (or it is not implemented).

authenticate(*password*)
Authenticate a tag with a *password*.

A tag that was once protected with a password requires authentication before write, potentially also read, operations may be performed. The *password* must be the same as the password provided to *protect()*. The return value indicates authentication success with `True` or `False`. For a tag that does not support authentication the return value is `None`.

exception nfc.tag.**TagCommandError**(*errno*)
Bases: `Exception`

The base class for exceptions that are raised when a tag command has not returned the expected result or a a lower stack error was raised.

The *errno* attribute holds a reason code for why the command has failed. Error numbers greater than zero indicate a tag type specific error from one of the exception classes derived from *TagCommandError* (per tag type module). Error numbers below and including zero indicate general errors:

```
nfc.tag.TIMEOUT_ERROR  => unrecoverable timeout error
nfc.tag.RECEIVE_ERROR  => unrecoverable transmission error
nfc.tag.PROTOCOL_ERROR => unrecoverable protocol error
```

The *TagCommandError* exception populates the *message* attribute of the general exception class with the appropriate error description.

errno
Holds the error reason code.

class nfc.tag.**TagEmulation**
Bases: `object`

Base class for tag emulation classes.

7.3.1 Type 1 Tag

exception nfc.tag.tt1.**Type1TagCommandError**(*errno*)
Bases: *nfc.tag.TagCommandError*

Type 1 Tag specific exceptions. Sets *errno* to one of:

1 - CHECKSUM_ERROR
2 - RESPONSE_ERROR
3 - WRITE_ERROR

class nfc.tag.tt1.**Type1Tag**(*clf*, *target*)
Bases: *nfc.tag.Tag*

Implementation of the NFC Forum Type 1 Tag Operation specification.

The NFC Forum Type 1 Tag is based on the ISO 14443 Type A technology for frame structure and anticollision (detection) commands, and the Innovision (now Broadcom) Jewel/Topaz commands for accessing the tag memory.

class NDEF(*tag*)
Bases: nfc.tag.NDEF

dump()
Returns the tag memory blocks as a list of formatted strings.

dump() iterates over all tag memory blocks (8 bytes each) from block zero until the physical end of memory and produces a list of strings that is intended for line by line printing. Multiple consecutive memory block of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory blocks present.

> **Warning:** For tags with more than 120 byte memory, the dump() method first overwrites the data block to verify that it is backed by physical memory, then restores the original data. This is necessary because Type 1 Tags do not indicate an error when reading beyond the physical memory space. Be cautious to not remove a tag from the reader when using dump() as otherwise your data may be corrupted.

protect(*password=None*, *read_protect=False*, *protect_from=0*)
The implementation of *nfc.tag.Tag.protect()* for a generic type 1 tag is limited to setting the NDEF data read-only for tags that are already NDEF formatted.

read_id()
Returns the 2 byte Header ROM and 4 byte UID.

read_all()
Returns the 2 byte Header ROM and all 120 byte static memory.

read_byte(*addr*)
Read a single byte from static memory area (blocks 0-14).

read_block(*block*)
Read an 8-byte data block at address (block * 8).

read_segment(*segment*)
Read one memory segment (128 byte).

write_byte(*addr*, *data*, *erase=True*)
Write a single byte to static memory area (blocks 0-14). The target byte is zero'd first if *erase* is True.

write_block(*block*, *data*, *erase=True*)
Write an 8-byte data block at address (block * 8). The target bytes are zero'd first if *erase* is True.

class nfc.tag.tt1_broadcom.**Topaz**(*clf*, *target*)
Bases: *nfc.tag.tt1.Type1Tag*

The Broadcom Topaz is a small memory tag that can hold up to 94 byte ndef message data.

dump()
Returns the tag memory blocks as a list of formatted strings.

dump() iterates over all tag memory blocks (8 bytes each) from block zero until the physical end of memory and produces a list of strings that is intended for line by line printing. Multiple consecutive memory block of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory blocks present.

> **Warning:** For tags with more than 120 byte memory, the dump() method first overwrites the data block to verify that it is backed by physical memory, then restores the original data. This is necessary because Type 1 Tags do not indicate an error when reading beyond the physical memory space. Be cautious to not remove a tag from the reader when using dump() as otherwise your data may be corrupted.

format(*version=None*, *wipe=None*)
Format a Topaz tag for NDEF use.

The implementation of *nfc.tag.Tag.format()* for a Topaz tag creates a capability container and an NDEF TLV with length zero. Data bytes of the NDEF data area are left untouched unless the wipe argument is set.

protect(*password=None*, *read_protect=False*, *protect_from=0*)
In addtion to *nfc.tag.tt1.Type1Tag.protect()* this method tries to set the lock bits to irreversibly protect the tag memory. However, it appears that tags sold have the lock bytes write protected, so this additional effort most likely doesn't have any effect.

class nfc.tag.tt1_broadcom.**Topaz512**(*clf*, *target*)
Bases: *nfc.tag.tt1.Type1Tag*

The Broadcom Topaz-512 is a memory enhanced version that can hold up to 462 byte ndef message data.

dump()
Returns the tag memory blocks as a list of formatted strings.

dump() iterates over all tag memory blocks (8 bytes each) from block zero until the physical end of memory and produces a list of strings that is intended for line by line printing. Multiple consecutive memory block of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory blocks present.

> **Warning:** For tags with more than 120 byte memory, the dump() method first overwrites the data block to verify that it is backed by physical memory, then restores the original data. This is necessary because Type 1 Tags do not indicate an error when reading beyond the physical memory space. Be cautious to not remove a tag from the reader when using dump() as otherwise your data may be corrupted.

format(*version=None*, *wipe=None*)
Format a Topaz-512 tag for NDEF use.

The implementation of *nfc.tag.Tag.format()* for a Topaz-512 tag creates a capability container, a Lock Control and a Memory Control TLV, and an NDEF TLV with length zero. Data bytes of the NDEF data area are left untouched unless the wipe argument is set.

protect(*password=None*, *read_protect=False*, *protect_from=0*)
In addtion to *nfc.tag.tt1.Type1Tag.protect()* this method tries to set the lock bits to irreversibly protect the tag memory. However, it appears that tags sold have the lock bytes write protected, so this additional effort most likely doesn't have any effect.

7.3.2 Type 2 Tag

exception nfc.tag.tt2.**Type2TagCommandError**(*errno*)
Bases: *nfc.tag.TagCommandError*

Type 2 Tag specific exceptions. Sets *errno* to one of:

1 - INVALID_SECTOR_ERROR
2 - INVALID_PAGE_ERROR
3 - INVALID_RESPONSE_ERROR

class nfc.tag.tt2.**Type2Tag**(*clf*, *target*)
Bases: *nfc.tag.Tag*

Implementation of the NFC Forum Type 2 Tag Operation specification.

The NFC Forum Type 2 Tag is based on the ISO 14443 Type A technology for frame structure and anticollision (detection) commands, and the NXP Mifare commands for accessing the tag memory.

class NDEF(*tag*)
Bases: nfc.tag.NDEF

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

format(*version=None*, *wipe=None*)
Erase the NDEF message on a Type 2 Tag.

The *format()* method will reset the length of the NDEF message on a type 2 tag to zero, thus the tag will appear to be empty. Additionally, if the *wipe* argument is set to some integer then *format()* will overwrite all user date that follows the NDEF message TLV with that integer (mod 256). If an NDEF message TLV is not present it will be created with a length of zero.

Despite it's name, the *format()* method can not format a blank tag to make it NDEF compatible. This is because the user data are of a type 2 tag can not be safely determined, also reading all memory pages until an error response yields only the total memory size which includes an undetermined number of special pages at the end of memory.

It is also not possible to change the NDEF mapping version, located in a one-time-programmable area of the tag memory.

protect(*password=None*, *read_protect=False*, *protect_from=0*)
Protect the tag against write access, i.e. make it read-only.

Type2Tag.protect() switches an NFC Forum Type 2 Tag to read-only state by setting all lock bits to 1. This operation can not be reversed. If the tag is not an NFC Forum Tag, i.e. it is not formatted with an NDEF Capability Container, the *protect()* method simply returns False.

A generic Type 2 Tag can not be protected with a password. If the *password* argument is provided, the *protect()* method does nothing else than return False. The *read_protect* and *protect_from* arguments are safely ignored.

read(*page*)
Send a READ command to retrieve data from the tag.

The *page* argument specifies the offset in multiples of 4 bytes (i.e. page number 1 will return bytes 4 to 19). The data returned is a byte array of length 16 or None if the block is outside the readable memory range.

Command execution errors raise *Type2TagCommandError*.

write(*page*, *data*)
Send a WRITE command to store data on the tag.

The *page* argument specifies the offset in multiples of 4 bytes. The *data* argument must be a string or bytearray of length 4.

Command execution errors raise *Type2TagCommandError*.

sector_select(*sector*)
Send a SECTOR_SELECT command to switch the 1K address sector.

The command is only send to the tag if the *sector* number is different from the currently selected sector number (set to 0 when the tag instance is created). If the command was successful, the currently selected sector number is updated and further *read()* and *write()* commands will be relative to that sector.

Command execution errors raise *Type2TagCommandError*.

transceive(*data*, *timeout=0.1*, *retries=2*)
Send a Type 2 Tag command and receive the response.

transceive() is a type 2 tag specific wrapper around the `nfc.ContactlessFrontend.exchange()` method. It can be used to send custom commands as a sequence of *data* bytes to the tag and receive the response data bytes. If *timeout* seconds pass without a response, the operation is aborted and *TagCommandError* raised with the TIMEOUT_ERROR error code.

Command execution errors raise *Type2TagCommandError*.

class nfc.tag.tt2.**Type2TagMemoryReader**(*tag*)
Bases: object

The memory reader provides a convenient way to read and write *Type2Tag* memory. Once instantiated with a proper type 2 *tag* object the tag memory can then be accessed as a linear sequence of bytes, without any considerations of sector or page boundaries. Modified bytes can be written to tag memory with *synchronize()*.

```
clf = nfc.ContactlessFrontend(...)
tag = clf.connect(rdwr={'on-connect': None})
if isinstance(tag, nfc.tag.tt2.Type2Tag):
    tag_memory = nfc.tag.tt2.Type2TagMemoryReader(tag)
    tag_memory[16:19] = [0x03, 0x00, 0xFE]
    tag_memory.synchronize()
```

synchronize()
Write pages that contain modified data back to tag memory.

class nfc.tag.tt2_nxp.**MifareUltralight**(*clf*, *target*)
Bases: *nfc.tag.tt2.Type2Tag*

Mifare Ultralight is a simple type 2 tag with no specific features. It can store up to 46 byte NDEF message data. This class does not do much more than to provide the known memory size.

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**MifareUltralightC**(*clf*, *target*)
Bases: *nfc.tag.tt2.Type2Tag*

Mifare Ultralight C provides more memory, to store up to 142 byte NDEF message data, and can be password protected.

class NDEF(*tag*)
Bases: nfc.tag.tt2.NDEF

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

protect(*password=None*, *read_protect=False*, *protect_from=0*)
Protect a Mifare Ultralight C Tag.

A Mifare Ultrlight C Tag can be provisioned with a custom password (or the default manufacturer key if the password is an empty string or bytearray).

A non-empty *password* must provide at least 128 bit key material, in other words it must be a string or bytearray of length 16 or more.

If *password* is not None, the first protected memory page can be specified with the *protect_from* integer argument. A memory page is 4 byte and the total number of pages is 48. A *protect_from* argument of 48 effectively disables memory protection. A *protect_from* argument of 3 protects all user data pages including the bitwise one-time-programmable page 3. Any value less than 3 or more than 48 is accepted but to the same effect as if 3 or 48 were specified. If effective protection starts at page 3 and the tag is formatted for NDEF, the *protect()* method does also modify the NDEF read/write capability byte.

If *password* is not None and *read_protect* is True then the tag memory content will also be protected against read access, i.e. successful authentication will be required to read protected pages.

The *protect()* method verifies a password change by authenticating with the new *password* after all modifications were made and returns the result of *authenticate()*.

> **Warning:** If protect is called without a password, the default Type 2 Tag protection method will set the lock bits to readonly. This process is not reversible.

authenticate(*password*)
Authenticate with a Mifare Ultralight C Tag.

autenticate() executes the Mifare Ultralight C mutual authentication protocol to verify that the *password* argument matches the key that is stored in the card. A new card key can be set with *protect()*.

The *password* argument must be a string with either 0 or at least 16 bytes. A zero length password string indicates that the factory default card key be used. From a password with 16 or more bytes the first 16 byte are taken as card key, remaining bytes are ignored. A password length between 1 and 15 generates a ValueError exception.

The authentication result is True if the password was confirmed and False if not.

class nfc.tag.tt2_nxp.**NTAG203**(*clf*, *target*)
Bases: *nfc.tag.tt2.Type2Tag*

The NTAG203 is a plain memory Tag with 144 bytes user data memory plus a 16-bit one-way counter. It does not have any security features beyond the standard lock bit mechanism that permanently disables write access.

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

protect (*password=None*, *read_protect=False*, *protect_from=0*)
Set lock bits to disable future memory modifications.

If *password* is None, all memory pages except the 16-bit counter in page 41 are protected by setting the relevant lock bits (note that lock bits can not be reset). If valid NDEF management data is found in page 4, protect() also sets the NDEF write flag to read-only.

The NTAG203 can not be password protected. If a *password* argument is provided, the protect() method always returns False.

class nfc.tag.tt2_nxp.**NTAG21x**(*clf*, *target*)
Bases: *nfc.tag.tt2.Type2Tag*

Base class for the NTAG21x family (210/212/213/215/216). The methods and attributes documented here are supported for all NTAG21x products.

All NTAG21x products support a simple password protection scheme that can be configured to restrict write as well as read access to memory starting from a selected page address. A factory programmed ECC signature allows to verify the tag unique identifier.

class NDEF(*tag*)
Bases: nfc.tag.tt2.NDEF

signature
The 32-byte ECC tag signature programmed at chip production. The signature is provided as a string and can only be read.

The signature attribute is always loaded from the tag when it is accessed, i.e. it is not cached. If communication with the tag fails for some reason the signature attribute is set to a 32-byte string of all zeros.

protect (*password=None*, *read_protect=False*, *protect_from=0*)
Set password protection or permanent lock bits.

If the *password* argument is None, all memory pages will be protected by setting the relevant lock bits (note that lock bits can not be reset). If valid NDEF management data is found, protect() also sets the NDEF write flag to read-only.

All Tags of the NTAG21x family can alternatively be protected by password. If a *password* argument is provided, the protect() method writes the first 4 byte of the *password* string into the Tag's password (PWD) memory bytes and the following 2 byte of the *password* string into the password acknowledge (PACK) memory bytes. Factory default values are used if the *password* argument is an empty string. Lock bits are not set for password protection.

The *read_protect* and *protect_from* arguments are only evaluated if *password* is not None. If *read_protect* is True, the memory protection bit (PROT) is set to require password verification also for reading of protected memory pages. The value of *protect_from* determines the first password protected memory page (one page is 4 byte) with the exception that the smallest set value is page 3 even if *protect_from* is smaller.

authenticate (*password*)
Authenticate with password to access protected memory.

An NTAG21x implements a simple password protection scheme. The reader proofs possession of a share secret by sending a 4-byte password and the tag proofs possession of a shared secret by returning a 2-byte password acknowledge. Because password and password acknowledge are transmitted in plain text special considerations should be given to under which conditions authentication is performed. If, for example, an attacker is able to mount a relay attack both secret values are easily lost.

The *password* argument must be a string of length zero or at least 6 byte characters. If the *password* length is zero, authentication is performed with factory default values. If the *password* contains at least 6 bytes, the first 4 byte are send to the tag as the password secret and the following 2 byte are compared against the password acknowledge that is received from the tag.

The authentication result is True if the password was confirmed and False if not.

class nfc.tag.tt2_nxp.**NTAG210**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.NTAG21x*

The NTAG210 provides 48 bytes user data memory, password protection, originality signature and a UID mirror function.

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**NTAG212**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.NTAG21x*

The NTAG212 provides 128 bytes user data memory, password protection, originality signature and a UID mirror function.

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**NTAG213**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.NTAG21x*

The NTAG213 provides 144 bytes user data memory, password protection, originality signature, a tag read counter and a mirror function for the tag unique identifier and the read counter.

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**NTAG215**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.NTAG21x*

The NTAG215 provides 504 bytes user data memory, password protection, originality signature, a tag read counter and a mirror function for the tag unique identifier and the read counter.

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**NTAG216**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.NTAG21x*

The NTAG216 provides 888 bytes user data memory, password protection, originality signature, a tag read counter and a mirror function for the tag unique identifier and the read counter.

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**MifareUltralightEV1**(*clf*, *target*, *product*)
Bases: *nfc.tag.tt2_nxp.NTAG21x*

Mifare Ultralight EV1

class nfc.tag.tt2_nxp.**MF0UL11**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.MifareUltralightEV1*

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**MF0ULH11**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.MifareUltralightEV1*

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**MF0UL21**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.MifareUltralightEV1*

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**MF0ULH21**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.MifareUltralightEV1*

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**NTAGI2C**(*clf*, *target*)
Bases: *nfc.tag.tt2.Type2Tag*

class nfc.tag.tt2_nxp.**NT3H1101**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.NTAGI2C*

NTAG I2C 1K.

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

class nfc.tag.tt2_nxp.**NT3H1201**(*clf*, *target*)
Bases: *nfc.tag.tt2_nxp.NTAGI2C*

NTAG I2C 2K.

dump()
Returns the tag memory pages as a list of formatted strings.

dump() iterates over all tag memory pages (4 bytes each) from page zero until an error response is received and produces a list of strings that is intended for line by line printing. Note that multiple consecutive memory pages of identical content may be reduced to fewer lines of output, so the number of lines returned does not necessarily correspond to the number of memory pages.

7.3.3 Type 3 Tag

exception nfc.tag.tt3.**Type3TagCommandError**(*errno*)
Bases: *nfc.tag.TagCommandError*

class nfc.tag.tt3.**ServiceCode**(*number*, *attribute*)
Bases: object

A service code provides access to a group of data blocks located on the card file system. A service code is a 16-bit structure composed of a 10-bit service number and a 6-bit service attribute. The service attribute determines the service type and whether authentication is required.

pack()
Pack the service code for transmission. Returns a 2 byte string.

classmethod unpack(*s*)
Unpack and return a ServiceCode from a byte string.

class nfc.tag.tt3.**BlockCode**(*number*, *access=0*, *service=0*)
Bases: object

A block code indicates a data block within a service. A block code is a 16-bit or 24-bit structure composed of a length bit (1b if the block number is less than 256), a 3-bit access mode, a 4-bit service list index and an 8-bit or 16-bit block number.

pack()
Pack the block code for transmission. Returns a 2-3 byte string.

class nfc.tag.tt3.**Type3Tag**(*clf*, *target*)
Bases: *nfc.tag.Tag*

Implementation of the NFC Forum Type 3 Tag specification.

The NFC Forum Type 3 Tag is based on the Sony FeliCa protocol and command specification. An NFC Forum compliant Type 3 Tag responds to a FeliCa polling command with system code 0x12FC and was configured to support service code 0x000B for NDEF data read and service code 0x0009 for NDEF data write (the latter may not be present if the tag is read-only) without encryption.

class NDEF(*tag*)
Bases: nfc.tag.NDEF

dump()
Read all data blocks of an NFC Forum Tag.

For an NFC Forum Tag (system code 0x12FC) *dump()* reads all data blocks from service 0x000B (NDEF read service) and returns a list of strings suitable for printing. The number of strings returned does not necessarily reflect the number of data blocks because a range of data blocks with equal content is reduced to fewer lines of output.

dump_service(*sc*)
Read all data blocks of a given service.

dump_service() reads all data blocks from the service with service code *sc* and returns a list of strings suitable for printing. The number of strings returned does not necessarily reflect the number of data blocks because a range of data blocks with equal content is reduced to fewer lines of output.

format(*version=None*, *wipe=None*)
Format and blank an NFC Forum Type 3 Tag.

A generic NFC Forum Type 3 Tag can be (re)formatted if it is in either one of blank, initialized or readwrite state. By formatting, all contents of the attribute information block is overwritten with values determined. The number of user data blocks is determined by reading all memory until an error response. Similarily, the maximum number of data block that can be read or written with a single command is determined by sending successively increased read and write commands. The current data length is set to zero. The NDEF mapping version is set to the latest known version number (1.0), unless the *version* argument is provided and it's major version number corresponds to one of the known major version numbers.

By default, no data other than the attribute block is modified. To overwrite user data the *wipe* argument must be set to an integer value. The lower 8 bits of that value are written to all data bytes that follow the attribute block.

polling(*system_code=65535*, *request_code=0*, *time_slots=0*)
Aquire and identify a card.

The Polling command is used to detect the Type 3 Tags in the field. It is also used for initialization and anti-collision.

The *system_code* identifies the card system to acquire. A card can have multiple systems. The first system that matches *system_code* will be activated. A value of 0xff for any of the two bytes works as a wildcard, thus 0xffff activates the very first system in the card. The card identification data returned are the Manufacture ID (IDm) and Manufacture Parameter (PMm).

The *request_code* tells the card whether it should return additional information. The default value 0 requests no additional information. Request code 1 means that the card shall also return the system code, so polling for system code 0xffff with request code 1 can be used to identify the first system on the card. Request code 2 asks for communication performance data, more precisely a bitmap of possible communication speeds. Not all cards provide that information.

The number of *time_slots* determines whether there's a chance to receive a response if multiple Type 3 Tags are in the field. For the reader the number of time slots determines the amount of time to wait for a response. Any Type 3 Tag in the field, i.e. powered by the field, will choose a random time slot to respond. With the default *time_slots* value 0 there will only be one time slot available for all responses and multiple responses would produce a collision. More time slots reduce the chance of collisions (but may result in an application working with a tag that was just accidentially close enough). Only specific values should be used for *time_slots*, those are 0, 1, 3, 7, and 15. Other values may produce unexpected results depending on the tag product.

polling() returns either the tuple (IDm, PMm) or the tuple (IDm, PMm, *additional information*) depending on the response lengt, all as bytearrays.

Command execution errors raise *TagCommandError*.

read_without_encryption (*service_list*, *block_list*)
Read data blocks from unencrypted services.

This method sends a Read Without Encryption command to the tag. The data blocks to read are indicated by a sequence of *BlockCode* objects in *block_list*. Each block code must reference a *ServiceCode* object from the iterable *service_list*. If any of the blocks and services do not exist, the tag will stop processing at that point and return a two byte error status. The status bytes become the *errno* value of the *TagCommandError* exception.

As an example, the following code reads block 5 from service 16 (service type 'random read-write w/o key') and blocks 0 to 1 from service 80 (service type 'random read-only w/o key'):

```
sc1 = nfc.tag.tt3.ServiceCode(16, 0x09)
sc2 = nfc.tag.tt3.ServiceCode(80, 0x0B)
bc1 = nfc.tag.tt3.BlockCode(5, service=0)
bc2 = nfc.tag.tt3.BlockCode(0, service=1)
bc3 = nfc.tag.tt3.BlockCode(1, service=1)
try:
    data = tag.read_without_encryption([sc1, sc2], [bc1, bc2, bc3])
except nfc.tag.TagCommandError as e:
    if e.errno > 0x00FF:
        print("the tag returned an error status")
    else:
        print("command failed with some other error")
```

Command execution errors raise *TagCommandError*.

read_from_ndef_service (**blocks*)
Read block data from an NDEF compatible tag.

This is a convinience method to read block data from a tag that has system code 0x12FC (NDEF). For other tags this method simply returns `None`. All arguments are block numbers to read. To actually pass a list of block numbers requires unpacking. The following example calls would have the same effect of reading 32 byte data from from blocks 1 and 8.:

```
data = tag.read_from_ndef_service(1, 8)
data = tag.read_from_ndef_service(*list(1, 8))
```

Command execution errors raise *TagCommandError*.

write_without_encryption (*service_list*, *block_list*, *data*)
Write data blocks to unencrypted services.

This method sends a Write Without Encryption command to the tag. The data blocks to overwrite are indicated by a sequence of *BlockCode* objects in the parameter *block_list*. Each block code must reference one of the *ServiceCode* objects in the iterable *service_list*. If any of the blocks or services do not exist, the tag will stop processing at that point and return a two byte error status. The status bytes become the *errno* value of the *TagCommandError* exception. The *data* to write must be a byte string or array of length `16 * len(block_list)`.

As an example, the following code writes `16 * "\xAA"` to block 5 of service 16, `16 * "\xBB"` to block 0 of service 80 and `16 * "\xCC"` to block 1 of service 80 (all services are writeable without key):

```
sc1 = nfc.tag.tt3.ServiceCode(16, 0x09)
sc2 = nfc.tag.tt3.ServiceCode(80, 0x09)
```

(continues on next page)

(continued from previous page)

```
bc1 = nfc.tag.tt3.BlockCode(5, service=0)
bc2 = nfc.tag.tt3.BlockCode(0, service=1)
bc3 = nfc.tag.tt3.BlockCode(1, service=1)
sc_list = [sc1, sc2]
bc_list = [bc1, bc2, bc3]
data = 16 * "\xAA" + 16 * "\xBB" + 16 * "\xCC"
try:
    data = tag.write_without_encryption(sc_list, bc_list, data)
except nfc.tag.TagCommandError as e:
    if e.errno > 0x00FF:
        print("the tag returned an error status")
    else:
        print("command failed with some other error")
```

Command execution errors raise *TagCommandError*.

write_to_ndef_service(*data*, **blocks*)
Write block data to an NDEF compatible tag.

This is a convinience method to write block data to a tag that has system code 0x12FC (NDEF). For other tags this method simply does nothing. The *data* to write must be a string or bytearray with length equal `16 * len(blocks)`. All parameters following *data* are interpreted as block numbers to write. To actually pass a list of block numbers requires unpacking. The following example calls would have the same effect of writing 32 byte zeros into blocks 1 and 8.:

```
tag.write_to_ndef_service(32 * "\0", 1, 8)
tag.write_to_ndef_service(32 * "\0", *list(1, 8))
```

Command execution errors raise *TagCommandError*.

send_cmd_recv_rsp(*cmd_code*, *cmd_data*, *timeout*, *send_idm=True*, *check_status=True*)
Send a command and receive a response.

This low level method sends an arbitrary command with the 8-bit integer *cmd_code*, followed by the captured tag identifier (IDm) if *send_idm* is `True` and the byte string or bytearray *cmd_data*. It then waits *timeout* seconds for a response, verifies that the response is correctly formatted and, if *check_status* is `True`, that the status flags do not indicate an error.

All errors raise a *TagCommandError* exception. Errors from response status flags produce an *errno* that is greater than 255, all other errors are below 256.

class nfc.tag.tt3.**Type3TagEmulation**(*clf*, *target*)
Bases: *nfc.tag.TagEmulation*

Framework for Type 3 Tag emulation.

class nfc.tag.tt3_sony.**FelicaStandard**(*clf*, *target*)
Bases: *nfc.tag.tt3.Type3Tag*

Standard FeliCa is a range of FeliCa OS based card products with a flexible file system that supports multiple applications and services on the same card. Services can individually be protected with a card key and all communication with protected services is encrypted.

dump()
Read all data blocks of an NFC Forum Tag.

For an NFC Forum Tag (system code 0x12FC) *dump()* reads all data blocks from service 0x000B (NDEF read service) and returns a list of strings suitable for printing. The number of strings returned does not necessarily reflect the number of data blocks because a range of data blocks with equal content is reduced to fewer lines of output.

request_service(*service_list*)
Verify existence of a service (or area) and get the key version.

Each service (or area) to verify must be given as a *ServiceCode* in the iterable *service_list*. The key versions are returned as a list of 16-bit integers, in the order requested. If a specified service (or area) does not exist, the key version will be 0xFFFF.

Command execution errors raise *TagCommandError*.

request_response()
Verify that a card is still present and get its operating mode.

The Request Response command returns the current operating state of the card. The operating state changes with the authentication process, a card is in Mode 0 after power-up or a Polling command, transitions to Mode 1 with Authentication1, to Mode 2 with Authentication2, and Mode 3 with any of the card issuance commands. The *request_response()* method returns the mode as an integer.

Command execution errors raise *TagCommandError*.

search_service_code(*service_index*)
Search for a service code that corresponds to an index.

The Search Service Code command provides access to the iterable list of services and areas within the activated system. The *service_index* argument may be any value from 0 to 0xffff. As long as there is a service or area found for a given *service_index*, the information returned is a tuple with either one or two 16-bit integer elements. Two integers are returned for an area definition, the first is the area code and the second is the largest possible service index for the area. One integer, the service code, is returned for a service definition. The return value is `None` if the *service_index* was not found.

For example, to print all services and areas of the active system:

```
for i in xrange(0x10000):
    area_or_service = tag.search_service_code(i)
    if area_or_service is None:
        break
    elif len(area_or_service) == 1:
        sc = area_or_service[0]
        print(nfc.tag.tt3.ServiceCode(sc >> 6, sc & 0x3f))
    elif len(area_or_service) == 2:
        area_code, area_last = area_or_service
        print("Area {0:04x}--{0:04x}".format(area_code, area_last))
```

Command execution errors raise *TagCommandError*.

request_system_code()
Return all system codes that are registered in the card.

A card has one or more system codes that correspond to logical partitions (systems). Each system has a system code that could be used in a polling command to activate that system. The system codes responded by the card are returned as a list of 16-bit integers.

```
for system_code in tag.request_system_code():
    print("System {0:04X}".format(system_code))
```

Command execution errors raise *TagCommandError*.

class nfc.tag.tt3_sony.**FelicaMobile**(*clf*, *target*)
Bases: *nfc.tag.tt3_sony.FelicaStandard*

Mobile FeliCa is a modification of FeliCa for use in mobile phones. This class does currently not implement anything specific beyond recognition of the Mobile FeliCa OS version.

class nfc.tag.tt3_sony.**FelicaLite**(*clf*, *target*)
Bases: *nfc.tag.tt3.Type3Tag*

FeliCa Lite is a version of FeliCa with simplified file system and security functions. The usable memory is 13 blocks (one block has 16 byte) plus a one block subtraction register. The tag can be configured with a card key to authenticate the tag and protect integrity of data reads.

class NDEF(*tag*)
Bases: nfc.tag.tt3.NDEF

dump()
Read all data blocks of an NFC Forum Tag.

For an NFC Forum Tag (system code 0x12FC) *dump()* reads all data blocks from service 0x000B (NDEF read service) and returns a list of strings suitable for printing. The number of strings returned does not necessarily reflect the number of data blocks because a range of data blocks with equal content is reduced to fewer lines of output.

protect(*password=None*, *read_protect=False*, *protect_from=0*)
Protect a FeliCa Lite Tag.

A FeliCa Lite Tag can be provisioned with a custom password (or the default manufacturer key if the password is an empty string or bytearray) to ensure that data retrieved by future read operations, after authentication, is genuine. Read protection is not supported.

A non-empty *password* must provide at least 128 bit key material, in other words it must be a string or bytearray of length 16 or more.

The memory unit for the value of *protect_from* is 16 byte, thus with protect_from=2 bytes 0 to 31 are not protected. If *protect_from* is zero (the default value) and the Tag has valid NDEF management data, the NDEF RW Flag is set to read only.

authenticate(*password*)
Authenticate a FeliCa Lite Tag.

A FeliCa Lite Tag is authenticated by a procedure that allows both the reader and the tag to calculate a session key from a random challenge send by the reader and a key that is securely stored on the tag and provided to *authenticate()* as the *password* argument. If the tag was protected with an earlier call to *protect()* then the same password should successfully authenticate.

After authentication the *read_with_mac()* method can be used to read data such that it can not be falsified on transmission.

format(*version=16*, *wipe=None*)
Format a FeliCa Lite Tag for NDEF.

read_without_mac(**blocks*)
Read a number of data blocks without integrity check.

This method accepts a variable number of integer arguments as the block numbers to read. The blocks are read with service code 0x000B (NDEF).

Tag command errors raise *TagCommandError*.

read_with_mac(**blocks*)
Read a number of data blocks with integrity check.

This method accepts a variable number of integer arguments as the block numbers to read. The blocks are read with service code 0x000B (NDEF). Along with the requested block data the tag returns a message authentication code that is verified before data is returned. If verification fails the return value of *read_with_mac()* is None.

A RuntimeError exception is raised if the tag was not authenticated before calling this method.

Tag command errors raise *TagCommandError*.

write_without_mac(*data*, *block*)
Write a data block without integrity check.

This is the standard write method for a FeliCa Lite. The 16-byte string or bytearray *data* is written to the numbered *block* in service 0x0009 (NDEF write service).

```
data = bytearray(range(16)) # 0x00, 0x01, ... 0x0F
try: tag.write_without_mac(data, 5) # write block 5
except nfc.tag.TagCommandError:
    print("something went wrong")
```

Tag command errors raise *TagCommandError*.

class nfc.tag.tt3_sony.**FelicaLiteS**(*clf*, *target*)
Bases: *nfc.tag.tt3_sony.FelicaLite*

FeliCa Lite-S is a version of FeliCa Lite with enhanced security functions. It provides mutual authentication were both the tag and the reader must demonstrate posession of the card key before data writes can be made. It is also possible to require mutual authentication for data reads.

class NDEF(*tag*)
Bases: nfc.tag.tt3_sony.NDEF

dump()
Read all data blocks of an NFC Forum Tag.

For an NFC Forum Tag (system code 0x12FC) *dump()* reads all data blocks from service 0x000B (NDEF read service) and returns a list of strings suitable for printing. The number of strings returned does not necessarily reflect the number of data blocks because a range of data blocks with equal content is reduced to fewer lines of output.

protect(*password=None*, *read_protect=False*, *protect_from=0*)
Protect a FeliCa Lite-S Tag.

A FeliCa Lite-S Tag can be write and read protected with a custom password (or the default manufacturer key if the password is an empty string or bytearray). Note that the *read_protect* flag is only evaluated when a *password* is provided.

A non-empty *password* must provide at least 128 bit key material, in other words it must be a string or bytearray of length 16 or more.

The memory unit for the value of *protect_from* is 16 byte, thus with protect_from=2 bytes 0 to 31 are not protected. If *protect_from* is zero (the default value) and the Tag has valid NDEF management data, the NDEF RW Flag is set to read only.

authenticate(*password*)
Mutually authenticate with a FeliCa Lite-S Tag.

FeliCa Lite-S supports enhanced security functions, one of them is the mutual authentication performed by this method. The first part of mutual authentication is to authenticate the tag with *FelicaLite.authenticate()*. If successful, the shared session key is used to generate the integrity check value for write operation to update a specific memory block. If that was successful then the tag is ensured that the reader has the correct card key.

After successful authentication the *read_with_mac()* and *write_with_mac()* methods can be used to read and write data such that it can not be falsified on transmission.

write_with_mac(*data*, *block*)
Write one data block with additional integrity check.

If prior to calling this method the tag was not authenticated, a `RuntimeError` exception is raised.

Command execution errors raise *`TagCommandError`*.

class nfc.tag.tt3_sony.**FelicaPlug**(*clf*, *target*)
Bases: *`nfc.tag.tt3.Type3Tag`*

FeliCa Plug is a contactless communication interface module for microcontrollers.

7.3.4 Type 4 Tag

exception nfc.tag.tt4.**Type4TagCommandError**(*errno*)
Bases: *`nfc.tag.TagCommandError`*

Type 4 Tag exception class. Beyond the generic error values from *`TagCommandError`* this class covers ISO 7816-4 response APDU error codes.

class nfc.tag.tt4.**Type4Tag**(*clf*, *target*)
Bases: *`nfc.tag.Tag`*

Implementation of the NFC Forum Type 4 Tag operation specification.

The NFC Forum Type 4 Tag is based on ISO/IEC 14443 DEP protocol for Type A and B modulation and uses ISO/IEC 7816-4 command and response APDUs.

class NDEF(*tag*)
Bases: `nfc.tag.NDEF`

dump()
Returns tag data as a list of formatted strings.

The *`dump()`* method provides useful output only for NDEF formatted Type 4 Tags. Each line that is returned contains a hexdump of 16 octets from the NDEF data file.

format(*version=None*, *wipe=None*)
Erase the NDEF message on a Type 4 Tag.

The *`format()`* method writes the length of the NDEF message on a Type 4 Tag to zero, thus the tag will appear to be empty. If the *wipe* argument is set to some integer then *`format()`* will also overwrite all user data with that integer (mod 256).

Despite it's name, the *`format()`* method can not format a blank tag to make it NDEF compatible; this requires proprietary information from the manufacturer.

transceive(*data*, *timeout=None*)
Transmit arbitrary data and receive the response.

This is a low level method to send arbitrary data to the tag. While it should almost always be better to use *`send_apdu()`* this is the only way to force a specific timeout value (which is otherwise derived from the Tag's answer to select). The *timeout* value is expected as a float specifying the seconds to wait.

send_apdu(*cla*, *ins*, *p1*, *p2*, *data=None*, *mrl=0*, *check_status=True*)
Send an ISO/IEC 7816-4 APDU to the Type 4 Tag.

The 4 byte APDU header (class, instruction, parameter 1 and 2) is constructed from the first four parameters (cla, ins, p1, p2) without interpretation. The byte string *data* argument represents the APDU command data field. It is encoded as a short or extended length field followed by the *data* bytes. The length field is not transmitted if *data* is None or an empty string. The maximum acceptable number of response data bytes is given with the max-response-length *mrl* argument. The value of *mrl* is transmitted as the 7816-4 APDU Le field after appropriate conversion.

By default, the response is returned as a byte array not including the status word, a *Type4TagCommandError* exception is raised for any status word other than 9000h. Response status verification can be disabled with *check_status* set to False, the byte array will then include the response status word at the last two positions.

Transmission errors always raise a *Type4TagCommandError* exception.

class nfc.tag.tt4.**Type4ATag**(*clf*, *target*)
Bases: *nfc.tag.tt4.Type4Tag*

class nfc.tag.tt4.**Type4BTag**(*clf*, *target*)
Bases: *nfc.tag.tt4.Type4Tag*

7.4 nfc.llcp

The nfc.llcp module implements the NFC Forum Logical Link Control Protocol (LLCP) specification and provides a socket interface to use the connection-less and connection-mode transport facilities of LLCP.

7.4.1 nfc.llcp.Socket

class nfc.llcp.**Socket**(*llc*, *sock_type*)
Bases: object

Create a new LLCP socket with the given socket type. The socket type should be one of:

- nfc.llcp.LOGICAL_DATA_LINK for best-effort communication using LLCP connection-less PDU exchange
- nfc.llcp.DATA_LINK_CONNECTION for reliable communication using LLCP connection-mode PDU exchange
- nfc.llcp.llc.RAW_ACCESS_POINT for unregulated LLCP PDU exchange (useful to implement test programs)

llc
The LogicalLinkController instance to which this socket belongs. This attribute is read-only.

resolve(*name*)
Resolve a service name into an address. This may involve conversation with the remote service discovery component if the name is hasn't yet been resolved. The return value is the service access point address for the service name bound at the remote device. The address value 0 indicates that the remote device does not have a service with the requested name. The address value 1 indicates that the remote device has a data link connection service with the requested name that can only be connected by service name. The return value is None when communication with the peer device terminated while waiting for a response.

setsockopt(*option*, *value*)
Set the value of the given socket option and return the current value which may have been corrected if it was out of bounds.

getsockopt(*option*)
Return the value of the given socket option.

bind(*address=None*)
Bind the socket to address. The socket must not already be bound. The address may be a service name string, a service access point number, or it may be omitted. If address is a well-known service name the socket will be bound to the corresponding service access point address, otherwise the socket will be bound to the next available service access point address between 16 and 31 (inclusively). If address is a number

between 32 and 63 (inclusively) the socket will be bound to that service access point address. If the address argument is omitted the socket will be bound to the next available service access point address between 32 and 63.

connect(*address*)
Connect to a remote socket at address. Address may be a service name string or a service access point number.

listen(*backlog*)
Mark a socket as a socket that will be used to accept incoming connection requests using accept(). The *backlog* defines the maximum length to which the queue of pending connections for the socket may grow. A backlog of zero disables queuing of connection requests.

accept()
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a new socket object usable to send and receive data on the connection.

send(*data*, *flags=0*)
Send data to the socket. The socket must be connected to a remote socket. Returns a boolean value that indicates success or failure. A false value is typically an indication that the socket or connection was closed.

sendto(*data*, *addr*, *flags=0*)
Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by addr. Returns a boolean value that indicates success or failure. Failure to send is generally an indication that the socket was closed.

recv()
Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data that may be returned is determined by the link or connection maximum information unit size.

recvfrom()
Receive data from the socket. The return value is a pair (bytes, address) where string is a string representing the data received and address is the address of the socket sending the data.

poll(*event*, *timeout=None*)
Wait for a socket event. Posssible *event* values are the strings "recv", "send" and "acks". Whent the timeout is present and not `None`, it should be a floating point number specifying the timeout for the operation in seconds (or fractions thereof). For "recv" or "send" the *poll()* method returns `True` if a next *recv()* or *send()* operation would be non-blocking. The "acks" event may only be used with a data-link-connection type socket; the call then returns `True` if the counter of received acknowledgements was greater than zero and decrements the counter by one.

getsockname()
Obtain the address to which the socket is bound. For an unbound socket the returned value is None.

getpeername()
Obtain the address of the peer connected on the socket. For an unconnected socket the returned value is None.

close()
Close the socket. All future operations on the socket object will fail. The remote end will receive no more data Sockets are automatically closed when the logical link controller terminates (gracefully or by link disruption). A connection-mode socket will attempt to disconnect the data link connection (if in connected state).

7.4.2 nfc.llcp.llc.LogicalLinkController

class nfc.llcp.llc.**LogicalLinkController**(***options*)
Bases: object

7.5 nfc.snep

The nfc.snep module implements the NFC Forum Simple NDEF Exchange Protocol (SNEP) specification and provides a server and client class for applications to easily send or receive SNEP messages.

7.5.1 nfc.snep.SnepServer

class nfc.snep.**SnepServer**(*llc*, *service_name='urn:nfc:sn:snep'*, *max_acceptable_length=1048576*, *recv_miu=1984*, *recv_buf=15*)
Bases: threading.Thread

NFC Forum Simple NDEF Exchange Protocol server

process_get_request(*ndef_message*)
Handle Get requests. This method should be overwritten by a subclass of SnepServer to customize it's behavior. The default implementation simply returns nfc.snep.NotImplemented.

process_put_request(*ndef_message*)
Process a SNEP Put request. This method should be overwritten by a subclass of SnepServer to customize it's behavior. The default implementation simply returns nfc.snep.Success.

7.5.2 nfc.snep.SnepClient

class nfc.snep.**SnepClient**(*llc*, *max_ndef_msg_recv_size=1024*)
Bases: object

Simple NDEF exchange protocol - client implementation

connect(*service_name*)
Connect to a SNEP server. This needs only be called to connect to a server other than the Default SNEP Server at urn:nfc:sn:snep or if the client wants to send multiple requests with a single connection.

close()
Close the data link connection with the SNEP server.

get_records(*records=None*, *timeout=1.0*)
Get NDEF message records from a SNEP Server.

New in version 0.13.

The ndef.Record list given by *records* is encoded as the request message octets input to *get_octets()*. The return value is an ndef.Record list decoded from the response message octets returned by *get_octets()*. Same as:

```
import ndef
send_octets = ndef.message_encoder(records)
rcvd_octets = snep_client.get_octets(send_octets, timeout)
records = list(ndef.message_decoder(rcvd_octets))
```

get_octets(*octets=None*, *timeout=1.0*)
Get NDEF message octets from a SNEP Server.

New in version 0.13.

If the client has not yet a data link connection with a SNEP Server, it temporarily connects to the default SNEP Server, sends the message octets, disconnects after the server response, and returns the received message octets.

put_records(*records*, *timeout=1.0*)
Send NDEF message records to a SNEP Server.

New in version 0.13.

The `ndef.Record` list given by *records* is encoded and then send via *put_octets()*. Same as:

```
import ndef
octets = ndef.message_encoder(records)
snep_client.put_octets(octets, timeout)
```

put_octets(*octets*, *timeout=1.0*)
Send NDEF message octets to a SNEP Server.

New in version 0.13.

If the client has not yet a data link connection with a SNEP Server, it temporarily connects to the default SNEP Server, sends the message octets and disconnects after the server response.

7.6 nfc.handover

The nfc.handover module implements the NFC Forum Connection Handover 1.2 protocol as a server and client class that simplify realization of handover selector and requester functionality.

7.6.1 nfc.handover.HandoverServer

class nfc.handover.**HandoverServer**(*llc*, *request_size_limit=65536*, *recv_miu=1984*, *recv_buf=15*)
Bases: `threading.Thread`

NFC Forum Connection Handover server

process_handover_request_message(*records*)
Process a handover request message. The *records* argument holds a list of `ndef.Record` objects decoded from the received handover request message octets, where the first record type is `urn:nfc:wkt:Hr`. The method returns a list of `ndef.Record` objects with the first record typ `urn:nfc:wkt:Hs`.

This method should be overwritten by a subclass to customize it's behavior. The default implementation returns a `ndef.HandoverSelectRecord` with version `1.2` and no alternative carriers.

7.6.2 nfc.handover.HandoverClient

class nfc.handover.**HandoverClient**(*llc*)
Bases: `object`

NFC Forum Connection Handover client

connect(*recv_miu=248*, *recv_buf=2*)
Connect to the remote handover server if available. Raises `nfc.llcp.ConnectRefused` if the remote device does not have a handover service or the service does not accept any more connections.

close()
Disconnect from the remote handover server.

send_records(*records*)
Send handover request message records to the remote server.

recv_records(*timeout=None*)
Receive a handover select message from the remote server.

Python Module Index

e

n

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W