nfcpy documentation

Release 1.0.3

Stephen Tiedemann

December 10, 2019

Contents

This documentation covers the '1.0.3' 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 2.6 or newer (but not Python 3)
- 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
- Run the standalone exe
- 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)
```

(continues on next page)

(continued from previous page)

```
...
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

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
```

(continues on next page)

(continued from previous page)

```
# 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.
# -----------------------------------------------------------------------------
"""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
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
            except AssertionError as error:
                print(str(error), file=sys.stderr)
                return 2

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

(continues on next page)

(continued from previous page)

```
            return target

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

    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)
```

(continues on next page)

(continued from previous page)

```
            return target
        except nfc.clf.CommunicationError:
            logging.error("communication failure after activation")


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

(continues on next page)

(continued from previous page)

```
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)))
```

5.5 rfstate.py

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.

**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:**
```

(continues on next page)

(continued from previous page)

```
  -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
                ("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")
```

(continues on next page)

(continued from previous page)

```
            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

7.2.1 Contactless Frontend

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

7.2.2 Technology Types

7.2.3 Exceptions

7.2.4 Driver Interface

7.2.5 Device Drivers

rcs380

pn531

pn532

pn533

rcs956

acr122

udp

7.3 nfc.tag

7.3.1 Type 1 Tag

7.3.2 Type 2 Tag

7.3.3 Type 3 Tag

7.3.4 Type 4 Tag

7.4 nfc.llcp

7.4.1 nfc.llcp.Socket

7.4.2 nfc.llcp.llc.LogicalLinkController

7.5 nfc.snep

7.5.1 nfc.snep.SnepServer

7.5.2 nfc.snep.SnepClient

7.6 nfc.handover

7.6.1 nfc.handover.HandoverServer

7.6.2 nfc.handover.HandoverClient

Index

Symbols

B

C

F

H

L

N

P

S

T

U