nfcpy documentation

Release 0.9.2

Stephen Tiedemann

January 04, 2017

Contents

Release v0.9.2

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 Python applications. The software is licensed under the EUPL.

To send a web link to a smartphone:

```
>>> import nfc, nfc.snep, threading
>>> connected = lambda llc: threading.Thread(target=llc.run).start()
>>> uri = nfc.ndef.Message(nfc.ndef.UriRecord("http://nfcpy.org"))
>>> clf = nfc.ContactlessFrontend('usb')
>>> llc = clf.connect(llcp={'on-connect': connected})
>>> nfc.snep.SnepClient(llc).put(uri)
True
>>> clf.close()
```

CHAPTER 1

Overview

1.1 Requirements

- Python 2.6 or newer but not Python 3.x
- pyUSB and libusb (for native USB readers)
- pySerial (for serial readers on COM or USB)

1.2 Supported Hardware

- Sony RC-S330/360/370/380
- SCM SCL-3710/11/12
- ACS ACR122U (version 2.xx)
- Arygon APPBUS
- Stollmann NFC Reader

Notes:

- All readers are tested to work with Ubuntu Linux. Less frequently some are tested to work on Windows (usually the SCL3711 and RC-S3xx). User feedback indicates that the readers seem to work on Mac. Readers with serial communication protocol have not yet been tested on Windows.
- The Sony RC-S380 is the only reader for which *nfcpy* currently supports tag emulation, more specifically Type 3 Tag emulation.
- The NXP PN53x can not properly handle Type 1 Tags with dynamic memory layout (Topaz 512) due to a firmware bug that does not allow READ-8 and WRITE-8 commands to be executed.
- The NXP PN531 chip does not support any Type 1 Tag command and is also not able to exchange Type 4 Tag commands if the ReadBinary and UpdateBinary commands exceed the length of a standard host controller frame (which may happen if the card sets ISO-DEP MIU as 256).
- The ACR122U is disabled as P2P Listener because the listen time can not be set less than 5 seconds. Also, because the reader has an MCU that controls a PN532 to implement the USB CCID protocol, it is generally less usable for NFC P2P communication due to the MCU interfering with settings made directly to the PN532.

1.3 Implementation Status

Specification	Status
TS NFC Digital Protocol 1.0	except Type B
TS NFC Activity 1.0	except Type B
TS Type 1 Tag Operation 1.1	implemented
TS Type 2 Tag Operation 1.1	implemented
TS Type 3 Tag Operation 1.1	implemented
TS Type 4 Tag Operation 1.0	implemented
TS Type 4 Tag Operation 2.0	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 1.0	not implemented
TS Logical Link Control Protocol 1.1	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://www.nfc-forum.org/specs/

CHAPTER 2

Getting started

2.1 Installation

1. Get the code

To get the latest development version:

```
$ sudo apt-get install bzr
$ cd <somedir>
$ bzr branch lp:nfcpy
```

This will download a branch of the nfcpy trunk repository from Canonical's Launchpad source code hosting platform into the local directory `<somedir>/trunk`.

For a Windows install the easiest is to download the Bazaar standalone installer from http://wiki.bazaar.canonical.com/WindowsDownloads and choose the *Typical Installation* that includes the *Bazaar Explorer GUI Application*. Start *Bazaar Explorer*, go to *Get project source from elsewhere* and create a local **branch** of `lp:nfcpy` into `C:/src/nfcpy` or some other directory of choice.

A release versions can be branched from the appropriate series, for example to grab the latest 0.0.x release.:

```
$ bzr branch lp:nfcpy/0.9
```

Tarballs of released versions are available for download at https://launchpad.net/nfcpy.

2. Install Python

Python is already installed on every Desktop Linux. Windows installers can be found at http://www.python.org/download/windows/. Make sure to choose a 2.x version, usually the latest, as *nfcpy* is not yet ported to Python 3.

3. Install libusb

The final piece needed is the USB library *libusb* and Python bindings. Once more this is dead easy for Linux where *libusb* is already available and the only step required is:

```
$ sudo apt-get install python-usb
```

To install libusb for Windows read the *Driver Installation* at http://www.libusb.org/wiki/windows_backend and use *Zadig.exe* to install *libusb-win32* for the contactless reader device (connect the reader and cancel the standard Windows install dialog, the device will be selectable in *Zadig*). The Python USB library can be downloaded as a zip file from http://sourceforge.net/projects/pyusb/ and installed with `python.exe setup.py install` from within the unzipped pyusb source code directory (add the full path to *python.exe* if it's not part of the search path).

4. Run example

A couple of example programs come with *nfcpy*. To see if the installation succeeded and the reader is working head over to the *nfcpy* directory and run the tagtool example:

```
$ python examples/tagtool.py show
```

Touch a compatible tag (NFC Forum Type 1-4) and the NDEF data should be printed. See tagtool.py for other options.

Note: Things may not immediately work on Linux for two reasons: The reader might be claimed by the Linux NFC subsystem available since Linux 3.1 and root privileges may be required to access the device. To prevent a reader being used by the NFC kernel driver add a blacklist entry in `'/etc/modprobe.d/'`, for example the following line works for the PN533 based SCL3711:

```
$ echo "blacklist pn533" | sudo tee -a /etc/modprobe.d/blacklist-nfc.conf
```

Root permissions are usually needed for the USB readers and `sudo python` is an easy fix, however not quite convinient and potentially dangerous. A better solution is to add a udev rule and make the reader accessible to a normal user, like the following rules would allow members of the *plugdev* group to access an SCL-3711 or RC-S380 if stored in `'/etc/udev/rules.d/nfcdev.rules'`.

```
SUBSYSTEM=="usb", ACTION=="add", ATTRS{idVendor}=="04e6", \
  ATTRS{idProduct}=="5591", GROUP="plugdev" # SCM SCL-3711
SUBSYSTEM=="usb", ACTION=="add", ATTRS{idVendor}=="054c", \
  ATTRS{idProduct}=="06c1", GROUP="plugdev" # Sony RC-S380
```

2.2 Open a reader

The main entrance to nfcpy is the *nfc.ContactlessFrontend* class. When initialized with a *path* argument it tries to locate and open a contacless reader connected at that location, which may be for example the first available reader on USB.

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend('usb')
>>> print(clf)
Sony RC-S360/SH on usb:002:005
```

For more control of where a reader may befound specifiy further details of the path string, for example *usb:002:005* to open the same reader as above, or *usb:002* to open the first available reader on USB bus number 2 (same numbers as shown by the *lsusb* command). The other way to specify a USB reader is by vendor and product ID, again by way of example *usb:054c:02e1* will most likely open the same reader as before if there's only one plugged in.

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend('usb:054c')
>>> print(clf)
Sony RC-S360/SH on usb:002:005
```

If you don't have an NFC reader at hand or just want to test your application logic a driver that carries NFC frames across a UDP/IP link might come handy.

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend('udp')
>>> print(clf)
Linux UDP/IP on udp:localhost:54321
```

Just to say for completeness, you can also omit the path argument and later open a reader using `ContactlessFrontend.open()`. The difference is that `open()` returns either `True` or `False` depending

on whether a reader was found whereas `ContactlessFrontend('...')` raises `IOError` if a reader was not found.

2.3 Read/write tags

With a reader opened the next step to get an NFC communication running is to use the `nfc.clf.ContactlessFrontend.connect()` method. We'll start with connecting to a tag (a contactless card), hopefully you have one and it's not a Mifare Classic. Currently supported are only NFC Forum Type 1, 2, 3 and 4 Tags.

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend('usb')
>>> clf.connect(rdwr={}) # now touch a tag and remove it
True
```

Although this doesn't look very exciting a lot has happened in the background. The tag was discovered and activated and it's data content read. Thereafter `nfc.clf.ContactlessFrontend.connect()` continued to check the presence of the tag until you removed it. The return value `True` tells us that it terminated normally and not due to a `KeyboardInterrupt` (in which case we've seen `False`). You can try this by either not touching or not removing the tag and press *Ctrl-C* while in `connect()`.

Obviously, as we've set the *rdwr* options as a dictionary, there must be something we can put into the dictionary to give us a bit more control. The most important option we can set is a callback funtion that will let us know when a tag got connected. It's famously called 'on-connect' and can be used like so:

```
>>> import nfc
>>> def connected(tag): print tag
...
>>> clf = nfc.ContactlessFrontend('usb')
>>> clf.connect(rdwr={'on-connect': connected}) # now touch a tag
Type3Tag IDm=01010501b00ac30b PMm=03014b024f4993ff SYS=12fc
<nfc.tag.tt3.Type3Tag object at 0x7f9e8302bfd0>
```

As expected our simple callback function does print some basic information about the tag, we see that it was an NFC Forum Type 3 Tag which has the system code 12FCh, a Manufacture ID and Manufacture Parameters. You should have noted that the connect() was not blocking until the tag was removed and that we've got an instance of class *nfc.tag.tt3.Type3Tag* returned. Both is because the callback function did return `None` (treated as `False` internally) and the connect() logic assumed that the caller want's to handle the tag presence check by itself or explicitely does not want to have that loop running. If we slightly modify the example you'll notice that again you have to remove the tag before connect() returns and the return value now is `True` (unless you press `Control-C` of course).

```
>>> import nfc
>>> def connected(tag): print tag; return True
...
>>> clf = nfc.ContactlessFrontend('usb')
>>> clf.connect(rdwr={'on-connect': connected}) # now touch a tag
Type3Tag IDm=01010501b00ac30b PMm=03014b024f4993ff SYS=12fc
True
```

Note: The generally recommended way for application logic on top of nfcpy is to use callback functions and not manually deal with the objects returned by connect(). But in the interactive Python interpreter it is sometimes just more convinient to do so. Tags are also quite friendly, they'll just wait indefinite time for you to send them a command, this is much different for LLCP and CARD mode where timing becomes critical. But more on that later.

Now that we've seen how to connect a tag, how do we get some data from it? If using the same tag as before, we've already learned by the system code 12FCh (which is specific for Type 3 Tags) that this tag should be capable to hold an NDEF message (NDEF is the NFC Forum Data Exchange Format and can be read and written with every NFC Forum compliant Tag). As *nfcpy* is supposed to make things easy, here is the small addition we need to get the NDEF message printed.

```
>>> import nfc
>>> with nfc.ContactlessFrontend('usb') as clf:
...     tag = clf.connect(rdwr={'on-connect': None}) # now touch a tag
...     print tag.ndef.message.pretty() if tag.ndef else "Sorry, no NDEF"
...
record 1
  type   = 'urn:nfc:wkt:Sp'
  name   = ''
  data   = '\xd1\x01\nU\x03nfcpy.org'
```

If the tag's attribute `ndef` is set we can simply read the ndef `message` attribute to get a fully parsed *nfc.ndef.Message* object, which in turn has a method to pretty print itself. It looks like this is a Smartposter message and probably links to the *nfcpy* website.

Note: We used two additional features to make our life easier and shorten typing. We've set the 'on-connect' callback to simply `None` instead of providing an actual function object that returns `None` (or `False` which would have the same effect). And we used `ContactlessFrontend` as a context manager, which means the *clf* it will be closed as soon as we leave the **with** clause.

Let's see if the Smartposter message is really referring to `nfcpy.org`. For that we'll need to know that NDEF parsers and generators are in the submodule `nfc.ndef`. And because it's easier to observe results step-by-step we'll not use the context manager mechanism but go plain. Just don't forget that you have either close the *clf* at the end of the example or leave the interpreter before trying the next example

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend('usb')
>>> tag = clf.connect(rdwr={'on-connect': None}) # now touch a tag
>>> if tag.ndef and tag.ndef.message.type == 'urn:nfc:wkt:Sp':
...     sp = nfc.ndef.SmartPosterRecord(tag.ndef.message[0])
...     print sp.pretty()
...
resource = http://nfcpy.org
action   = default
```

There are a few things to note. First, we went one step further in attribute the hierarchy and discovered the message type. An *nfc.ndef.Message* is a sequence of *nfc.ndef.Record* objects, each having a *type*, a *name* and a *data* member. The *type* and *name* of the first record are simply mapped to the *type* and *name* of the message itself as that usually sets the processing context for the remaining records. Second, we grab the first record by index 0 without any check for an index error. Of course may that be safe due to the initial check on message type (which turns to the first record type) and we'd expect something else to be there if the message is empty. But it's also safe because the *tag.ndef.message* will **always** hold a valid *Message*, just that it be a message with one empty record (*type*, *name* and *data* will all be empty strings) if the NDEF tag does not contain actual NDEF data or the data is corrupted.

Now as the final piece of this section let us improve the Smartposter a little bit. Usually a Smartposter should have a URI that links to the resource and a title to help humans understand what the link points to. We omit all the safety check, so please be sure to touch the same tag as before and not switch to a Mifare Classic.

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend('usb')
>>> tag = clf.connect(rdwr={'on-connect': None}) # now touch the tag
>>> sp = nfc.ndef.SmartPosterRecord('http://nfcpy.org')
```

```
>>> sp.title = "Python module for near field communication"
>>> tag.ndef.message = nfc.ndef.Message(sp)
>>> print nfc.ndef.SmartPosterRecord(tag.ndef.message[0]).pretty()
resource  = http://nfcpy.org
title[en] = Python module for near field communication
action    = default
```

It happend, you've destroyed your overly expensive contactless tag. Sorry I was joking, except for the "overly expensive" part (they should really become cheaper). But the tag, if nothing crashed, has now slightly different content and it all happend in the sixth line were the new message got assigned to the `tag.ndef.message` attribute. In that line it was immediately written to the tag and the internal copy (the old data) invalidated. The last line then caused the message to be read back from the tag and finally printed.

Note: The *nfc.ndef* module has a lot more functionality than could be covered in this short introduction, feel free to read the API documentation as well as the *NFC Data Exchange Format* tutorial to learn how *nfcpy* maps to the concepts of the NDEF specification.

2.4 Pretend a card

How do we get *nfcpy* to be a card? Supply `card` options to *nfc.ContactlessFrontend.connect()*.

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend('usb')
>>> print clf.connect(card={})
None
```

Guess you've noticed that something was going wrong. Unlike when reading a card (or tag) the `clf.connect()` call returns immediately and the result we're getting is `None`. This is because there exists no sensible default behavior that can be applied when working as a tag, we need to be explicit about the technology we want to use (for a tag reader it just makes sense to look for all technologies and tag types). So we choose a technology and supply that as the 'targets' option.

```
>>> import nfc
>>> clf = nfc.ContactlessFrontend('usb')
>>> nfcf_idm = bytearray.fromhex('03FEFFE011223344')
>>> nfcf_pmm = bytearray.fromhex('01E0000000FFFF00')
>>> nfcf_sys = bytearray.fromhex('12FC')
>>> target = nfc.clf.TTF(br=212, idm=nfcf_idm, pmm=nfcf_pmm, sys=nfcf_sys)
>>> clf.connect(card={'targets': [target]}) # touch a reader
True
```

Note: It is time to talk about the limitations. As of writing, *nfcpy* supports tag emulation only for NFC Forum Type 3 Tag and requires a Sony RC-S380 contactless frontend. The only alternative to an RC-S380 is to use the UDP driver that simulates NFC communication over UDP/IP. To use the UDP driver initialize ContactlessFrontend with the string `udp` and use `examples/tagtool.py --device udp` as card reader.

You can read the tag we've created for example with the excellent NXP Tag Info app available for free in the Android app store. It will tell you that this is a *FeliCa Plug RC-S926* tag (because we said that with the first two bytes of the *IDm*) and if you switch over to the TECH view there'll be the *IDm*, *PMm* and *System Code* we've set.

Note: Depending on your Android device it will be more or less difficult to get a stable reading, it seems to depend

much on the phone's NFC chip and driver. Generally the Google Nexus 4 and 10 work pretty well and the same should be true for the Samsung S4 as those are having the same chip. Other phones can be a little bitchy.

The NXP Tag Info app tells us that there's no NDEF partition on it, so let's fix that. It's unfortunately now going to be a bit more code and you probably want to copy it, so the following is not showing the interpreter prompt.

```
import nfc
clf = nfc.ContactlessFrontend('usb')
nfcf_idm = bytearray.fromhex('03FEFFE011223344')
nfcf_pmm = bytearray.fromhex('01E0000000FFFF00')
nfcf_sys = bytearray.fromhex('12FC')
target = nfc.clf.TTF(br=212, idm=nfcf_idm, pmm=nfcf_pmm, sys=nfcf_sys)

attr = nfc.tag.tt3.NdefAttributeData()
attr.version, attr.nbr, attr.nbw = '1.0', 12, 8
attr.capacity, attr.writeable = 1024, True
ndef_data_area = str(attr) + bytearray(attr.capacity)

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 connected(tag, cmd):
    tag.add_service(0x0009, ndef_read, ndef_write)
    tag.add_service(0x000B, ndef_read, lambda: False)
    return True

while clf.connect(card={'targets': [target], 'on-connect': connected}): pass
```

We've now got a fully functional NFC Forum Type 3 Tag. If, for example, you have the NXP Tag Writer app installed, start to write something to the card, touch again to read it back, and so on. Finally, press `Ctrl-C` to stop the card working.

Note: Other card commands can be realized by running the basic *receive command* and *send response* loop as part of the application logic, for example as part of the `on-connect` callback function with a `False` value returned at the end. The loop requires a bit of exception checking and must handle unknown command, check out *`nfc.ContactlessFrontend.connect()`* in `nfc/clf.py` for something to start with.

2.5 Work with a peer

The best part of NFC comes when the limitations of a single master controlling a poor 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 established again by calling `ContactlessFrontend.connect()` with a set of options, this time they go with the argument `llcp`.

Note: The example code in this section assumes that you have an Android phone to use as peer device. If that is not the case you can either use readers that are supported by *nfcpy* and start `examples/snep-test-server.py --loop` before diving into the examples or use the UDP driver to work without a hardware. You'll then start `examples/snep-test-server.py --loop --device udp` first and initalize `ContactlessFrontend()` with the path string `'udp'` instead of `'usb'`.

Here's the shortest code fragment we can use to get an LLCP link running.

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

Depending on your reader and the phone you may have had to explicitely move both out of proximity to see `True` printed after connect or it may just have happened. That is simply because the device connect phase may have seen unstable communication and `connect` returns after one activation/deactivation.

Note: In the contactless world it can not be really distinguished whether deactivation was intentional deactivation or because of broken communication. A broken communication is just the normal case when a user removes the device.

Remember that `connect()` returns `True` (or something that evaluates `True` in a boolean expression) when returning normally and the pattern is clear: We just need to call `connect()` in an endless loop until a `KeyboardInterrupt` exception is raised (with `Ctrl-C` or send by an external program)

```
>>> import nfc
>>> clf = ContactlessFrontend('usb')
>>> while clf.connect(llcp={}): pass
...
>>> clf.close()
```

Now we've got LLCP running but there's still not much we can do with it. But same as for the other modes we can add a callback function for the `on-connect` event. This function will receive as it's single argument the *`LogicalLinkController`* instance that controls the LLCP link.

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

The callback function is the place where we to start LLCP client and server applications but it is important to treat it like an interrupt, that means application code must be started in a separate thread and the callback return immediately. The reason is that in order to keep the LLCP link alive and receive or dispatch LLC protocol data units (PDUs) the *`LogicalLinkController`* must run a service loop and `connect()` is using the calling thread's context for that. When using the interactive interpreter this is less convinient as we'd have to change the callback code when going

further with the tutorial, so remember that if the callback returns `False` or `None` then `connect()` will not do the housekeeping stuff but return immediately and give us the callback parameters.

```
>>> import nfc, threading
>>> clf = nfc.ContactlessFrontend('usb')
>>> connected = lambda llc: threading.Thread(target=llc.run).start()
>>> llc = clf.connect(llcp={'on-connect': connected})
>>> print llc
LLC: Local(MIU=128, LTO=100ms) Remote(MIU=1024, LTO=500ms)
>>> clf.close()
```

Application code is not supposed to work directly with the *llc* object but it's one of the parameters we need to create a *nfc.llcp.Socket* for the actual communication. The other parameter we need to supply is the socket type, 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.

So what can we do next with the Android phone? It happens that every modern NFC phone on the market has a so called SNEP Default Server running that we can play with. The acronym SNEP stands for the NFC Forum Simple NDEF Exchange Protocol and the SNEP Default Server is a service that must be available on every NFC Forum certified device. Though many phones are not yet certified, a SNEP default server is built into stock Android as part of the Android Beam feature. Because SNEP messages are exchanged over an LLCP data link connection we'll first have to create a connection-mode socket, then determine the address of the SNEP server, connect to the server and finally send some data.

```
>>> import nfc, threading
>>> clf = nfc.ContactlessFrontend('usb')
>>> connected = lambda llc: threading.Thread(target=llc.run).start()
>>> llc = clf.connect(llcp={'on-connect': connected})
>>> socket = nfc.llcp.Socket(llc, nfc.llcp.DATA_LINK_CONNECTION)
>>> addr = socket.resolve('urn:nfc:sn:snep')
>>> addr
4
>>> socket.connect(addr)
>>> msg = nfc.ndef.Message(nfc.ndef.SmartPosterRecord("http://nfcpy.org"))
>>> str(msg)
'\xd1\x02\x0eSp\xd1\x01\nU\x03nfcpy.org'
>>> hex(len(str(msg)))
'0x13'
>>> socket.send("\x10\x02\x00\x00\x00\x13" + str(msg))
>>> socket.recv()
'\x10\x81\x00\x00\x00\x00'
>>> socket.close()
>>> clf.close()
```

If your phone has an Internet connection you should now see that the Internet browser has opened the http://nfcpy.org web page. In Android terminology we've *beamed*.

Just for the purpose of demonstration the example did resolve the SNEP default server's service name into an address value first. But both the service name `urn:nfc:sn:snep` and the address 4 are well-known values defined in the NFC Forum Assigned Numbers Register and so we've could have directly connect to address 4. And because it is also possible to use a service name as an address we've could have gone without the reolve step at all. So all of the following calls would have brought us the same effect.

```
>>> socket.connect( socket.resolve('urn:nfc:sn:snep') )
>>> socket.connect( 'urn:nfc:sn:snep' )
>>> socket.connect( 4 )
```

As it is a primary goal of *nfcpy* to make life as simple as possible, there is no need to mess around with binary strings. The *nfc.snep.SnepClient* does all the things needed, just import *nfc.snep* to have it available.

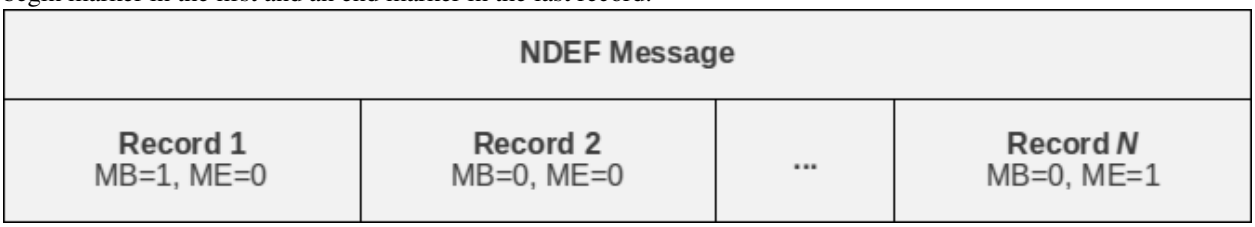
```
>>> import nfc, nfc.snep, threading
>>> clf = nfc.ContactlessFrontend('usb')
>>> connected = lambda llc: threading.Thread(target=llc.run).start()
>>> llc = clf.connect(llcp={'on-connect': connected})
>>> link = nfc.ndef.UriRecord("http://nfcpy.org")
>>> snep = nfc.snep.SnepClient(llc)
>>> snep.put(nfc.ndef.Message(link))
True
>>> clf.close()
```

The *nfc.llcp* module documentation contains more information on LLCP and the *nfc.llcp.Socket* API.

CHAPTER 3

NFC Data Exchange Format

NDEF (NFC Data Exchange Format) is a binary message format to exchange application-defined payloads between NFC Forum Devices or to store payloads on an NFC Forum Tag. A payload is described by a type, a length and an optional identifer encoded in an NDEF record structure. An NDEF message is a sequence of NDEF records with a begin marker in the first and an end marker in the last record.



NDEF decoding and encoding is provided by the *nfc.ndef* module.

```
>>> import nfc.ndef
```

3.1 Parsing NDEF

An *nfc.ndef.Message* class can be initialized with an NDEF message octet string to parse that data into the sequence of NDEF records framed by the begin and end marker of the first and last record. Each NDEF record is represented by an *nfc.ndef.Record* object accessible through indexing or iteration over the *nfc.ndef.Message* object.

```
>>> import nfc.ndef
>>> message = nfc.ndef.Message(b'\xD1\x01\x0ET\x02enHello World')
>>> message
nfc.ndef.Message([nfc.ndef.Record('urn:nfc:wkt:T', '', '\x02enHello World')])
>>> len(message)
1
>>> message[0]
nfc.ndef.Record('urn:nfc:wkt:T', '', '\x02enHello World')
>>> for record in message:
>>>     record.type, record.name, record.data
>>>
('urn:nfc:wkt:T', '', '\x02enHello World')
```

An NDEF record carries three parameters for describing its payload: the payload length, the payload type, and an optional payload identifier. The *nfc.ndef.Record.data* attribute provides access to the payload and the payload length is obtained by `len()`. The *nfc.ndef.Record.name* attribute holds the payload identifier and is an empty

string if no identifer was present in the NDEF date. The *nfc.ndef.Record.type* identifies the type of the payload as a combination of the NDEF Type Name Format (TNF) field and the type name itself.

Empty (TNF 0)

An *Empty* record type (expressed as a zero-length string) indicates that there is no type or payload associated with this record. Encoding a record of this type will exclude the name (*payload identifier*) and data (*payload*) contents. This type can be used whenever an empty record is needed; for example, to terminate an NDEF message in cases where there is no payload defined by the user application.

NFC Forum Well Known Type (TNF 1)

An *NFC Forum Well Known Type* is a URN as defined by **RFC 2141**, with the namespace identifier (NID) "nfc". The Namespace Specific String (NSS) of the NFC Well Known Type URN is prefixed with "wkt:". When encoded in an NDEF message, the Well Known Type is written as a relative-URI construct (cf. **RFC 3986**), omitting the NID and the "wkt:" -prefix. For example, the type "urn:nfc:wkt:T" will be encoded as TNF 1, TYPE "T".

Media-type as defined in RFC 2046 (TNF 2)

A *media-type* follows the media-type BNF construct defined by **RFC 2046**. Records that carry a payload with an existing, registered media type should use this record type. Note that the record type indicates the type of the payload; it does not refer to a MIME message that contains an entity of the given type. For example, the media type 'image/jpeg' indicates that the payload is an image in JPEG format using JFIF encoding as defined by **RFC 2046**.

Absolute URI as defined in RFC 3986 (TNF 3)

An *absolute-URI* follows the absolute-URI BNF construct defined by **RFC 3986**. This type can be used for message types that are defined by URIs. For example, records that carry a payload with an XML-based message type may use the XML namespace identifier of the root element as the record type, like a SOAP/1.1 message may be represented by the URI 'http://schemas.xmlsoap.org/soap/envelope/'.

NFC Forum External Type (TNF 4)

An *NFC Forum External Type* is a URN as defined by **RFC 2141**, with the namespace identifier (NID) "nfc". The Namespace Specific String (NSS) of the NFC Well Known Type URN is prefixed with "ext:". When encoded in an NDEF message, the External Type is written as a relative-URI construct (cf. **RFC 3986**), omitting the NID and the "ext:" -prefix. For example, the type "urn:nfc:ext:nfcpy.org:T" will be encoded as TNF 4, TYPE "nfcpy.org:T".

Unknown (TNF 5)

An *Unknown* record type (expressed by the string "unknown") indicates that the type of the payload is unknown, similar to the "application/octet-stream" media type.

Unchanged (TNF 6)

An *Unchanged* record type (expressed by the string "unchanged") is used in middle record chunks and the terminating record chunk used in chunked payloads. This type is not allowed in any other record.

```
>>> import nfc.ndef
>>> message = nfc.ndef.Message('\xD0\x00\x00')
>>> nfc.ndef.Message('\xD0\x00\x00')[0].type
''
>>> nfc.ndef.Message('\xD1\x01\x00T')[0].type
'urn:nfc:wkt:T'
>>> nfc.ndef.Message('\xD2\x0A\x00text/plain')[0].type
'text/plain'
>>> nfc.ndef.Message('\xD3\x16\x00http://example.org/dtd')[0].type
'http://example.org/dtd'
>>> nfc.ndef.Message('\xD4\x10\x00example.org:Text')[0].type
```

```
'urn:nfc:ext:example.org:Text'
>>> nfc.ndef.Message('\xD5\x00\x00')[0].type
'unknown'
>>> nfc.ndef.Message('\xD6\x00\x00')[0].type
'unchanged'
```

The type and name of the first record, by convention, provide the processing context and identification not only for the first record but for the whole NDEF message. The *nfc.ndef.Message.type* and *nfc.ndef.Message.name* attributes map to the type and anme attributes of the first record in the message.

```
>>> message = nfc.ndef.Message(b'\xD1\x01\x0ET\x02enHello World')
>>> message.type, message.name
('urn:nfc:wkt:T', '')
```

If invalid or insufficient data is provided to to the NDEF message parser, an `nfc.ndef.FormatError` or `nfc.ndef.LengthError` is raised.

```
>>> try: nfc.ndef.Message('\xD0\x01\x00')
... except nfc.ndef.LengthError as e: print e
...
insufficient data to parse
>>> try: nfc.ndef.Message('\xD0\x01\x00T')
... except nfc.ndef.FormatError as e: print e
...
ndef type name format 0 doesn't allow a type string
```

3.2 Creating NDEF

An *nfc.ndef.Record* class can be initialized with an NDEF

To build NDEF messages use the *nfc.ndef.Record* class to create records and instantiate an *nfc.ndef.Message* object with the records as arguments.

```
>>> import nfc.ndef
>>> record1 = nfc.ndef.Record("urn:nfc:wkt:T", "id1", "\x02enHello World!")
>>> record2 = nfc.ndef.Record("urn:nfc:wkt:T", "id2", "\x02deHallo Welt!")
>>> message = nfc.ndef.Message(record1, record2)
```

The *nfc.ndef.Message* class also accepts a list of records as a single argument and it is possible to *nfc.ndef.Message.append()* records or *nfc.ndef.Message.extend()* a message with a list of records.

```
>>> message = nfc.ndef.Message()
>>> message.append(record1)
>>> message.extend([record2, record3])
```

The serialized form of an *nfc.ndef.Message* object is produced with `str()`.

```
>>> message = nfc.ndef.Message(record1, record2)
>>> str(message)
'\x99\x01\x0f\x03Tid1\x02enHello World!Y\x01\x0e\x03Tid2\x02deHallo Welt!'
```

3.3 Specific Records

3.3.1 Text Record

```
>>> import nfc.ndef
>>> record = nfc.ndef.TextRecord("Hello World!")
>>> print record.pretty()
text     = Hello World!
language = en
encoding = UTF-8
```

3.3.2 Uri Record

```
>>> import nfc.ndef
>>> record = nfc.ndef.UriRecord("http://nfcpy.org")
>>> print record.pretty()
uri = http://nfcpy.org
```

3.3.3 Smart Poster Record

```
>>> import nfc.ndef
>>> uri = "https://launchpad.net/nfcpy"
>>> record = nfc.ndef.SmartPosterRecord(uri)
>>> record.title = "Python module for near field communication"
>>> record.title['de'] = "Python Modul für Nahfeldkommunikation"
>>> print record.pretty()
resource  = https://launchpad.net/nfcpy
title[de] = Python Modul für Nahfeldkommunikation
title[en] = Python module for near field communication
action    = default
```

CHAPTER 4

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", address=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(clf, 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(clf, 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()))
    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()
    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 5

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.

5.1 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 put(self, ndef_message):
        print "client has put an NDEF message"
        print ndef_message.pretty()
        return nfc.snep.Success

def startup(clf, 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)
```

5.2 Using SNEP Put

Sending an NDEF message to the *Default SNEP Server* is easily done with an instance of *nfc.snep.SnepClient* and is basically to call *nfc.snep.SnepClient.put()* with the message to send. 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 run the LLCP link.

```
import nfc
import nfc.snep
import threading

def send_ndef_message(llc):
    sp = nfc.ndef.SmartPosterRecord('http://nfcpy.org', title='nfcpy home')
    snep = nfc.snep.SnepClient(llc)
    snep.put( nfc.ndef.Message(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 may just do nothing.

```
import nfc
import nfc.snep
import threading

server = None

def send_ndef_message(llc):
    sp = nfc.ndef.SmartPosterRecord('http://nfcpy.org', title='nfcpy home')
    snep = nfc.snep.SnepClient(llc)
    snep.put( nfc.ndef.Message(sp) )

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})
```

5.3 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 = nfc.ndef.Message(nfc.ndef.Record())
        service_name = "urn:nfc:xsn:nfcpy.org:x-snep"
        nfc.snep.SnepServer.__init__(self, llc, service_name, 2048)

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

    def get(self, acceptable_length, ndef_message):
        print "client requests an NDEF message"
        if ((ndef_message.type == '' and ndef_message.name == '') or
            ((ndef_message.type == self.ndef_message.type) and
             (ndef_message.name == self.ndef_message.name))):
            if len(str(self.ndef_message)) > acceptable_length:
                return nfc.snep.ExcessData
            return self.ndef_message
        return nfc.snep.NotFound

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. It is a arbitrary restriction of the *nfc.snep.SnepClient* that this parameter can only be set when the object is created; the SNEP protocol would allow it to be different for every GET request but unless there's demand for such flexibility that won't change.

```
import nfc
import nfc.snep
import threading

def send_ndef_message(llc):
    sp = nfc.ndef.SmartPosterRecord('http://nfcpy.org', title='nfcpy home')
    snep = nfc.snep.SnepClient(llc, max_ndef_msg_recv_size=2048)
    snep.connect("urn:nfc:xsn:nfcpy.org:x-snep")
    snep.put( nfc.ndef.Message(sp) )

    print "*** get whatever the server has ***"
    print snep.get().pretty()

    print "*** get a smart poster with no name ***"
    r = nfc.ndef.Record(record_type="urn:nfc:wkt:Sp", record_name="")
    print snep.get( nfc.ndef.Message(r) ).pretty()

    print "*** get something that isn't there ***"
```

```
    r = nfc.ndef.Record(record_type="urn:nfc:wkt:Uri")
    try:
        snep.get( nfc.ndef.Message(r) )
    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 6

Example Programs

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

ndeftool.py Generate or inspect or reorganize NDEF data.

beam.py Exchange NDEF data with a smartphone.

6.1 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
```


6.1.1 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. Only relevant for reader/writer mode.

`-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 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. The tag type must be specified. The only currently supported tag type it **tt3**.

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

format tt1

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

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

```
$ 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 STR
: Type 3 Tag NDEF mapping version number, specified as a version string with minor and major number separated by a single dot character. Both major and minor version numbers must be in range `0<=N<=15`. The default value is `"1.1"`.

--nbr INT
: Type 3 Tag attribute block *Nbr* field value, the number of blocks that can be read at once. Must be in range `0<=INT<=255`. If this option is not specified the automatically detected value is written.

--nbw INT
: Type 3 Tag attribute block *Nbw* field value, the number of blocks that can be written at once. Must be in range `0<=INT<=255`. If this option is not specified the automatically detected value is written.

--max INT
: Type 3 Tag attribute block *Nmaxb* field value, which is the maximum number of blocks available for NDEF data. Must be in range `0<=INT<=255`. If this option is not specified the automatically detected value is written.

--rfu INT
: Type 3 Tag attribute block *reserved* field value. Must be in range `0<=INT<=255`. The default value is 0.

--wf INT
: Type 3 Tag attribute block *WriteF* field value. Must be in range `0<=INT<=255`. The default value is 0.

--rw INT
: Type 3 Tag attribute block *RW Flag* field value. Must be in range `0<=INT<=255`. The default value is 1.

--len INT
: Type 3 Tag attribute block *Ln* field value that specifies the actual size of the NDEF data stored. Must be in range `0<=INT<=16777215`. The default value is 0.

--crs INT
: Type 3 Tag attribute block *Checksum* field value. Must be in range `0<=INT<=65535`. If this option is not specified the automatically computed checksum is written.

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 maximum size for NDEF data. Depending on the tag type this may be rounded to the nearest multiple of the tag storage granularity.

-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.

6.1.3 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
```

6.2 ndeftool.py

The **ndeftool** intends to be a swiss army knife for working with NDEF data.

```
$ ndeftool.py [-h] [-v] [-d] {print,make,pack,split,cat} ...
```


6.2.1 Options

-v
: Print informational messages.

-d
: Print debug information.

6.2.2 Commands

print

The **print** command decodes and prints the content of an NDEF message. The message data may be in raw binary or hexadecimal string format and is read from *message-file* or standard input if *message-file* is not provided.:

```
$ ndeftool.py print [-h|--help] [message]
```

make

The **make** command allows generating specific NDEF messages. The type of message is determined by a further sub-command:

- **make smartposter** - creates a smartposter record
- **make wificfg** - creates a WiFi Configuration record
- **make wifipwd** - creates a WiFi Password record
- **make btcfg** - creates a Bluetooth out-of-band record

make smartposter

The **make smartposter** command creates a smartposter message for the uniform resource identifier *reference*:

```
$ ndeftool.py make smartposter [-h|--help] [options] reference
```

Options:

-t `titlespec`
: Add a smart poster title. The *titlespec* consists of an ISO/IANA language code, a ":" as separator, and the title string. The language code is optional and defaults to "en". The separator may then also be omitted unless the title string itself contains a colon. Multiple `-t` options may be present for different languages.

-i `iconfile`
: Add a smart poster icon. The *iconfile* must be an existing and readable image file for which a mime type is registered. Multiple `-i` options may be present for different image types.

-a `actionstring`
: Set the smart poster action. Valid action strings are "default" (default action of the receiving device), "exec" (send SMS, launch browser, call phone number), "save" (store SMS in INBOX, bookmark hyperlink, save phone number in contacts), and "edit".

-o `output-file`
: Write message data to *output-file* (default is write to standard output). The `-o` option also switches the output format to raw bytes versus the hexadecimal string written to stdout.

make wificfg

The **make wificfg** command creates a configuration token for the WiFi network with SSID *network-name*. Without further options this command creates configuration data for an open network:

```
$ ndeftool.py make wificfg [-h|--help] [options] network-name
```

Options:

--key network-key
Set the *network-key* for a secured WiFi network. The security method is set to WPA2-Personal.

--mixed-mode
With this option set the security method is set to also include the older WPA-Personal standard.

--mac mac-address
The MAC address of the device for which the credential was generated. Without the `--mac` option the broadcast MAC "ff:ff:ff:ff:ff:ff" is used to indicate that the credential is not device specific.

--shareable
Set this option if the network configuration may be shared with other devices.

-o output-file
Write message data to *output-file* (default is write to standard output). The `-o` option also switches the output format to raw bytes versus the hexadecimal string written to stdout.

--hs
Encapsulate the Wifi Configuration record into a Handover Select Message. The carrier power state will set to 'unknown'.

--active
Generate a Handover Select message with the WiFi carrier power state set to 'active'. This option is mutually exclusive with the `--inactive` and `--activating` options.

--inactive
Generate a Handover Select message with the WiFi carrier power state set to 'inactive'. This option is mutually exclusive with the `--active` and `--activating` options.

--activating
Generate a Handover Select message with the WiFi carrier power state set to 'activating'. This option is mutually exclusive with the `--active` and `--inactive` options.

make wifipwd

The **make wifipwd** command creates a password token for the WiFi Protected Setup registration protocol, signed with the first 160 bits of SHA-256 hash of the enrollee's public key in *public-key-file*.:

```
$ ndeftool.py make wificfg [-h|--help] [options] public-key-file
```

Options:

-p device-password
A 16 - 32 octet long device password. If the `-p` option is not given a 32 octet long random device password is generated.

-i password-id
An arbitrary value between 0x0010 and 0xFFFF that serves as an identifier for the device password. If the `-i` option is not given a random password identifier is generated.

-o output-file
Write message data to *output-file* (default is write to standard output). The `-o` option also switches the output format to raw bytes versus the hexadecimal string written to stdout.

make btcfg

The **make btcfg** command creates an out-of-band configuration record for a Bluetooth device.:

```
$ ndeftool.py make btcfg [-h|--help] [options] device-address
```

Options:

-c class-of-device
: The 24 class of device/service bits as a string of '0' and '1' characters, with the most significant bit left.

-n name-of-device
: The user friendly name of the device.

-s service-class
: A service class implemented by the device. A service class may be specified by description or as a 128-bit UUID string (for example, "00001108-0000-1000-8000-00805f9b34fb" would indicate "Printing"). Textual descriptions are evaluated case insensitive and must then match one of the following:

  'Handsfree Audio Gateway', 'PnP Information', 'Message Access Server', 'ESDP UPNP IP PAN', 'HDP Source', 'Generic Networking', 'Message Notification Server', 'Browse Group Descriptor', 'NAP', 'A/V Remote Control Target', 'Basic Imaging Profile', 'Generic File Transfer', 'Message Access Profile', 'Generic Telephony', 'Basic Printing', 'Intercom', 'HCR Print', 'Dialup Networking', 'Advanced Audio Distribution', 'Printing Status', 'OBEX File Transfer', 'Handsfree', 'Hardcopy Cable Replacement', 'Imaging Responder', 'Phonebook Access - PSE', 'ESDP UPNP IP LAP', 'IrMC Sync', 'Cordless Telephony', 'LAN Access Using PPP', 'OBEX Object Push', 'Video Source', 'Audio Source', 'Human Interface Device', 'Video Sink', 'Reflected UI', 'ESDP UPNP L2CAP', 'Service Discovery Server', 'HDP Sink', 'Direct Printing Reference', 'Serial Port', 'SIM Access', 'Imaging Referenced Objects', 'UPNP Service', 'A/V Remote Control Controller', 'HCR Scan', 'Headset - HS', 'UPNP IP Service', 'IrMC Sync Command', 'GNSS', 'Headset', 'WAP Client', 'Imaging Automatic Archive', 'Phonebook Access', 'Fax', 'Generic Audio', 'Audio Sink', 'GNSS Server', 'A/V Remote Control', 'Video Distribution', 'WAP', 'Common ISDN Access', 'Direct Printing', 'GN', 'PANU', 'Phonebook Access - PCE', 'Headset - Audio Gateway (AG)', 'Reference Printing', 'HDP'

-o output-file
: Write message data to *output-file* (default is write to standard output). The `-o` option also switches the output format to raw bytes versus the hexadecimal string written to stdout.

--hs
: Encapsulate the Bluetooth Configuration record into a Handover Select Message. The carrier power state will set to 'unknown' unless one of the options *–active*, *–inactive* or *–activating* is given.

--active
: Generate a Handover Select message with the Bluetooth carrier power state set to 'active'. This option is mutually exclusive with the `--inactive` and `--activating` options.

--inactive
: Generate a Handover Select message with the Bluetooth carrier power state set to 'inactive'. This option is mutually exclusive with the `--active` and `--activating` options.

--activating
: Generate a Handover Select message with the Bluetooth carrier power state set to 'activating'. This option is mutually exclusive with the `--active` and `--inactive` options.

pack

The **pack** command converts a file into an NDEF record with both message begin and end flag set to 1. If the `-t` option is not given the record type is guessed from the file content using the mimetypes module. The record name is by default set to the name of the file being converted, unless data is read from stdin in which case the record name is not encoded.

If a file mime type is `text/plain` it will be encoded as an NDEF Text Record (type `urn:nfc:wkt:T`) if `-t` is not set. The text record language is guessed from the file content if the Python module `guess_language` is installed,

otherwise set to English.

```
$ ndeftool.py pack [-h|--help] [options] FILE
```

Options:

-t record-type
: Set the record type to *record-type* (the default is to guess it from the file mime type).

-n record-name
: Set the record identifier to *record-name* (the default is to use the file path name).

-o output-file
: Write message data to *output-file* (default is write to standard output). The -o option also switches the output format to raw bytes versus the hexadecimal string written to stdout.

split

The **split** command separates an an NDEF message into individual records. If data is read from a file, records are written as binary data into individual files with file names constructed from the input file base name, a hyphen followed by a three digit number and the input file name extension. If data is read from stdin, records are written to stdout as individual lines of hexadecimal strings.

```
$ ndeftool.py split [-h|--help] [options] message-file
```

Options:

--keep-message-flags
: Do not reset the record's message begin and end flags but leave tem as found in the input message data.

cat

The **cat** command concatenates records into a single message.

```
$ ndeftool.py cat [-h|--help] record-file [record-file ...]
```

Options:

-o output-file
: Write message data to *output-file* (default is write to standard output). The -o option also switches the output format to raw bytes versus the hexadecimal string written to stdout.

6.2.3 Examples

To build a smartposter that points to the nfcpy documentation page:

```
$ ndeftool.py make smartposter http://nfcpy.org/docs
d102135370d1010f55036e666370792e6f72672f646f6373
```

The output can be made readable with the ndeftool print command:

```
$ ndeftool.py make smartposter http://nfcpy.org/docs | ndeftool.py print
Smartposter Record
  resource = http://nfcpy.org/docs
  action   = default
```

To get the smartposter as raw bytes specify an output file:

```
$ ndeftool.py make smartposter http://nfcpy.org/docs -o sp_nfcpy_docs.ndef
```

Here's a more complex example setting multi-language smartposter title, icons and a non-default action:

```
$ ndeftool.py make smartposter http://nfcpy.org/docs -t "nfcpy documentation" -t "de:nfcpy Dokumentat
```

It is sometimes helpful to have an NDEF message of specific length where the payload consists of monotonically increasing byte values:

```
$ python -c "import sys; sys.stdout.write(bytearray([x % 256 for x in xrange(1024-6)]))" | ndeftool.p
```

6.3 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]
```


6.3.1 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.

6.3.2 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 NFC Data Exchange Format 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.

6.3.3 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
```

CHAPTER 7

Interoperability Tests

7.1 Logical Link Control Protocol

7.1.1 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.

7.1.2 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.

7.2 Simple NDEF Exchange Protocol

7.2.1 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.

7.2.2 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.

7.3 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.

7.3.1 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.

7.3.2 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.

7.4 Personal Health Device Communication

7.4.1 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. Only relevant for reader/writer mode.

`-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.

7.4.2 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.

7.4.3 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.

7.5 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.

7.5.1 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 read
$ ./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.p
$ ./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.p
$ ./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.p
$ ./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.p
$ ./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 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.p
$ ./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.p
$ ./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.p
$ ./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.p
$ ./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 8

Module Reference

8.1 nfc

8.1.1 nfc.ContactlessFrontend

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

The contactless frontend is the main interface class for working with contactless reader devices. A reader device may be opened when an instance is created by providing the *path* argument, see *nfc.ContactlessFrontend.open()* for how it must be constructed.

The initializer method raises IOError(errno.ENODEV) if a path is specified but no no reader are found.

open(*path*)

Open a contactless reader device identified by *path*.

Parameters **path** – search path for contactless reader

Returns **True** if reader was found and activated

Path specification:

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.

close()

Close the contacless reader device.

connect (***options*)

Connect with a contactless target or become connected as a contactless target. The calling thread is blocked until a single activation and deactivation has completed or a callback function supplied as the keyword argument `terminate` returned True. The result of the terminate function also applies to the loop run after activation, so the example below will make *connect()* return after 10 seconds from either waiting for a peer device or when connected.

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

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

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

Reader/Writer Options

'targets': sequence A list of target specifications with each target of either type *TTA*, *TTB*, or *TTF*. A default set is choosen if 'targets' is not provided.

'on-startup': function A function that will be called with the list of targets (from 'targets') to search for. Must return a list of targets or `None`. Only the targets returned are finally considered.

'on-connect': function A function object that will be called with an activated `Tag` object.

```
>>> import nfc
>>> def connected(tag):
...     print tag
...     return True
...
>>> clf = nfc.ContactlessFrontend()
>>> clf.connect(rdwr={'on-connect': connected})
Type3Tag IDm=01010501b00ac30b PMm=03014b024f4993ff SYS=12fc
True
```

Peer To Peer Options

'on-startup': function A function that is called before an attempt is made to establish peer to peer communication. The function receives the initialized *LogicalLinkController* instance as parameter, which may then be used to allocate and bind communication sockets for service applications. The return value must be either the *LogicalLinkController* instance or `None` to effectively remove llcp from the options considered.

'on-connect': function A function that is be called when peer to peer communication was established. The function receives the connected *LogicalLinkController* instance as parameter, which may then be used to allocate communication sockets with `socket()` and spawn working threads to perform communication. The callback must return more or less immediately with `True` unless the logical link controller run loop is handled within the callback.

'role': string Defines which role the local LLC shall take for the data exchange protocol activation. Possible values are 'initiator' and 'target'. The default is to alternate between both roles until communication is established.

'miu': integer Defines the maximum information unit size that will be supported and announced to the remote LLC. The default value is 128.

'lto': integer Defines the link timeout value (in milliseconds) that will be announced to the remote LLC. The default value is 100 milliseconds.

'agf': boolean Defines if the local LLC performs PDU aggregation and may thus send Aggregated Frame (AGF) PDUs to the remote LLC. The dafault is to use aggregation.

```
>>> import nfc
>>> import threading
>>> def worker(socket):
...     socket.sendto("Hi there!", address=16)
...     socket.close()
...
>>> def connected(llc):
...     socket = llc.socket(nfc.llcp.LOGICAL_DATA_LINK)
...     threading.Thread(target=worker, args=(socket,)).start()
...     return True
...
>>> clf = nfc.ContactlessFrontend()
>>> clf.connect(llcp={'on-connect': connected})
```

Card Emulation Options

'targets': sequence A list of target specifications with each target of either type *TTA*, *TTB*, or *TTF*. The list of targets is processed sequentially. Defaults to an empty list.

'on-startup': function A function that will be called with the list of targets (from 'targets') to emulate. Must return a list of one target choosen or `None`.

'on-connect': function A function that will be called with an activated `TagEmulation` instance as first parameter and the first command received as the second parameter.

'on-release': function A function that will be called when the activated tag has been released by it's Initiator, basically that is when the tag has been removed from the Initiator's RF field.

'timeout': integer The timeout in seconds to wait for for each target to become initialized. The default value is 1 second.

```
>>> import nfc
>>>
>>> def connected(tag, command):
...     print tag
...     print str(command).encode("hex")
...
>>> clf = nfc.ContactlessFrontend()
>>> idm = bytearray.fromhex("01010501b00ac30b")
>>> pmm = bytearray.fromhex("03014b024f4993ff")
>>> sys = bytearray.fromhex("12fc")
>>> target = nfc.clf.TTF(212, idm, pmm, sys)
>>> clf.connect(card={'targets': [target], 'on-connect': connected})
Type3TagEmulation IDm=01010501b00ac30b PMm=03014b024f4993ff SYS=12fc
100601010501b00ac30b010b00018000
True
```

Connect returns `None` if no options were to execute, `False` if interrupted by a `KeyboardInterrupt`, or `True` if terminated normally and the 'on-connect' callback function had returned `True`. If the 'on-connect' callback had returned `False` the return value of connect() is the same parameters as were provided to the callback function.

Connect raises `IOError(errno.ENODEV)` if called before a contactless reader was opened.

sense(*targets*, ***kwargs*)
Send discovery and activation requests to find a target. Targets is a list of target specifications (TTA, TTB, TTF). Not all readers may support all possible target types. The return value is an activated target with a possibly updated specification (bitrate) or None.

Additional keyword arguments are driver specific.

Note: This is a direct interface to the driver and not needed if *connect()* is used.

listen(*target*, *timeout*)
Listen for *timeout* seconds to become initialized as a *target*. The *target* must be one of *nfc.clf.TTA*, *nfc.clf.TTB*, *nfc.clf.TTF*, or *nfc.clf.DEP* (note that target type support depends on the hardware capabilities). The return value is `None` if *timeout* elapsed without activation or a tuple (target, command) where target is the activated target (which may differ from the requested target, see below) and command is the first command received from the initiator.

If an activated target is returned, the target type and attributes may differ from the *target* requested. This is especically true if activation as a *nfc.clf.DEP* target is requested but the contactless frontend does not have a hardware implementation of the data exchange protocol and returns a *nfc.clf.TTA* or *nfc.clf.TTF* target instead.

Note: This is a direct interface to the driver and not needed if *connect()* is used.

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

Note: This is a direct interface to the driver and not needed if *connect()* is used.

set_communication_mode(*brm*, ***kwargs*)
Set the hardware communication mode. The effect of calling this method depends on the hardware support, some drivers may purposely ignore this function. If supported, the parameter *brm* specifies the communication mode to choose as a string composed of the bitrate and modulation type, for example '212F' shall switch to 212 kbps Type F communication. Other communication parameters may be changed with optional keyword arguments. Currently implemented by the RC-S380 driver are the parameters 'add-crc' and 'check-crc' when running as initator. It is possible to set *brm* to an empty string if bitrate and modulation shall not be changed but only optional parameters executed.

Note: This is a direct interface to the driver and not needed if *connect()* is used.

class `nfc.clf.`**TTA**(*br=None*, *cfg=None*, *uid=None*, *ats=None*)
Represents a Type A target. The integer *br* is the bitrate. The bytearray *cf* is the two byte SENS_RES data plus the one byte SEL_RES data for a tag type 1/4 tag. The bytearray *uid* is the target UID. The bytearray *ats* is the answer to select data of a type 4 tag if the chipset does activation as part of discovery.

class `nfc.clf.`**TTB**(*br=None*)
Represents a Type B target. The integer *br* is the bitrate. Type B targets are not yet supported in nfcpy, for the

simple reason that no cards for testing are available.

class nfc.clf.**TTF**(*br=None*, *idm=None*, *pmm=None*, *sys=None*)
Represents a Type F target. The integer *br* is the bitrate. The bytearray *idm* is the 8 byte manufacture id. The bytearray *pmm* is the 8 byte manufacture parameter. The bytearray *sys* is the 2 byte system code.

class nfc.clf.**DEP**(*br=None*, *gb=None*)
Represents a DEP target. The integer *br* is the bitrate. The bytearray *gb* is the ATR general bytes.

8.2 nfc.tag

8.2.1 nfc.tag.tt1.Type1Tag

class nfc.tag.tt1.**Type1Tag**

is_present
Returns True if the tag is still within communication range.

read_id()
Read header rom and all static memory bytes (blocks 0-14).

read_all()
Read header rom and all static memory bytes (blocks 0-14).

read_byte(*addr*)
Read a single byte from static memory area (blocks 0-14).

write_byte(*addr*, *byte*, *erase=True*)
Write a single byte to static memory area (blocks 0-14). The target byte is zero'd first if 'erase' is True (default).

read_block(*block*)
Read an 8-byte data block at address (block * 8).

write_block(*block*, *data*, *erase=True*)
Write an 8-byte data block at address (block * 8). The target bytes are zero'd first if 'erase' is True (default).

8.2.2 nfc.tag.tt2.Type2Tag

class nfc.tag.tt2.**Type2Tag**

is_present
Returns True if the tag is still within communication range.

read(*block*)
Read 16-byte of data from the tag. The *block* argument specifies the offset in multiples of 4 bytes (i.e. block number 1 will return bytes 4 to 19). The data returned is a byte array of length 16.

write(*block*, *data*)
Write 4-byte of data to the tag. The *block* argument specifies the offset in multiples of 4 bytes. The *data* argument must be a string or bytearray of length 4.

8.2.3 nfc.tag.tt3.Type3Tag

class nfc.tag.tt3.**Type3Tag**

is_present
True if the tag is still within communication range.

poll(*system_code*)
Send the polling command to recognize a system on the card. The *system_code* may be specified as a short integer or as a string or bytearray of length 2. The return value is the tuple of the two bytearrays (idm, pmm) if the requested system is present or the tuple (None, None) if not.

read(*blocks*, *service=11*)
Read service data blocks from tag. The *service* argument is the tag type 3 service code to use, 0x000b for reading NDEF. The *blocks* argument holds a list of integers representing the block numbers to read. The data is returned as a character string.

write(*data*, *blocks*, *service=9*)
Write service data blocks to tag. The *service* argument is the tag type 3 service code to use, 0x0009 for writing NDEF. The *blocks* argument holds a list of integers representing the block numbers to write. The *data* argument must be a character string with length equal to the number of blocks times 16.

8.2.4 nfc.tag.tt4.Type4Tag

class nfc.tag.tt4.**Type4Tag**

is_present
True if the tag is still within communication range.

select_file(*p1*, *p2*, *data*, *expected_response_length=None*)
Select a file or directory with parameters defined in ISO/IEC 7816-4

read_binary(*offset*, *count*)
Read *count* bytes from selected file starting at *offset*

update_binary(*offset*, *data*)
Write *data* bytes to selected file starting at *offset*

8.3 nfc.ndef

Support for decoding and encoding of NFC Data Exchange Format (NDEF) records and messages.

8.3.1 nfc.ndef.Message

class nfc.ndef.**Message**(**args*)

Wraps a sequence of NDEF records and provides methods for appending, inserting and indexing. Instantiation accepts a variable number of positional arguments. A call without argument produces a Message object with no records. A single str or bytearray argument is parsed as NDEF message bytes. A single list or tuple of *nfc.ndef.Record* objects produces a Message with those records in order. One or more *nfc.ndef.Record* arguments produce a Message with those records in order.

```
>>> nfc.ndef.Message(b'\x10\x00\x00')      # NDEF data bytes
>>> nfc.ndef.Message(bytearray([16,0,0])) # NDEF data bytes
>>> nfc.ndef.Message([record1, record2])  # list of records
>>> nfc.ndef.Message(record1, record2)    # two record args
```

append(*record*)

Add a record to the end of the message. The *record* argument must be an instance of *nfc.ndef.Record*.

extend(*records*)

Extend the message by appending all the records in the given list. The *records* argument must be a sequence of *nfc.ndef.Record* elements.

insert(*i*, *record*)

Insert a record at the given position. The first argument *i* is the index of the record before which to insert, so message.insert(0, record) inserts at the front of the message, and message.insert(len(message), record) is equivalent to message.append(record). The second argument *record* must be an instance of *nfc.ndef.Record*.

pop(*i=-1*)

Remove the record at the given position *i* in the message, and return it. If no position is specified, message.pop() removes and returns the last item.

type

The message type. Corresponds to the record type of the first record in the message. None if the message has no records. This attribute is read-only.

name

The message name. Corresponds to the record name of the first record in the message. None if the message has no records. This attribute is read-only.

pretty()
Returns a message representation that might be considered pretty-printable.

8.3.2 nfc.ndef.Record

class nfc.ndef.**Record**(*record_type=None*, *record_name=None*, *data=None*)
Wraps an NDEF record and provides getting and setting of the record type name (*type*), record identifier (*name*) and record payload (*data*).

Parameters

- **record_type** – NDEF record type name
- **record_name** – NDEF record identifier
- **data** – NDEF record payload or NDEF record data

All arguments accept a `str` or `bytearray` object.

Interpretation of the *data* argument depends on the presence of *record_type* and *record_name*. If any of the *record_type* or *record_name* argument is present, the *data* argument is interpreted as the record payload and copied to *data*. If none of the *record_type* or *record_name* argument are present, the *data* argument is interpreted as a NDEF record bytes (NDEF header and payload) and parsed.

The *record_type* argument combines the NDEF TNF (Type Name Format) and NDEF TYPE information into a single string. The TNF values 0, 5 and 6 are expressed by the strings ‘’, ‘unknown’ and ‘unchanged’. For TNF values 2 and 4 the *record_type* is the prefix ‘urn:nfc:wkt:’ and ‘urn:nfc:ext:’, respectively, followed by the NDEF TYPE string. TNF values 2 and 3 are not distinguished by regular expressions matching the either the media-type format ‘type-name/subtype-name’ or absolute URI format ‘scheme:hier-part’

```
>>> nfc.ndef.Record('urn:nfc:wkt:T', 'id', b'}enHello World')
>>> nfc.ndef.Record('urn:nfc:wkt:T', data=b'}enHello World')
>>> nfc.ndef.Record(data=b'Ñ{T}enHello World')
```

type
The record type. A string that matches the empty string ‘’, or the string ‘unknown’, or the string ‘unchanged’, or starts with ‘urn:nfc:wkt:’, or starts with ‘urn:nfc:ext:’, or matches the mime-type format, or matches the absolute-URI format.

name
The record identifier as an octet string. Any type that can be coverted into a sequence of characters in range(0,256) can be assigned.

data
The record payload as an octet string. Any type that can be coverted into a sequence of characters in range(0,256) can be assigned.

pretty(*indent=0*)
Returns a string with a formatted representation that might be considered pretty-printable. The optional argument *indent* specifies the amount of indentation added for each level of output.

class nfc.ndef.record.**RecordList**(*iterable=()*)
Bases: `list`

A specialized list type that only accepts *Record* objects.

8.3.3 nfc.ndef.TextRecord

class nfc.ndef.**TextRecord**(*text=None*, *language='en'*, *encoding='UTF-8'*)
Bases: nfc.ndef.record.Record

Wraps an NDEF Text record and provides access to the *encoding*, *language* and actual *text* content.

Parameters

- **text** – Text string or *nfc.ndef.Record* object
- **language** – ISO/IANA language code string
- **encoding** – Text encoding in binary NDEF

The *text* argument may alternatively supply an instance of class *nfc.ndef.Record*. Initialization is then done by parsing the record payload. If the record type does not match 'urn:nfc:wkt:T' a ValueError exception is raised.

```
>>> nfc.ndef.TextRecord(nfc.ndef.Record())
>>> nfc.ndef.TextRecord("English UTF-8 encoded")
>>> nfc.ndef.TextRecord("Deutsch UTF-8", language="de")
>>> nfc.ndef.TextRecord("English UTF-16", encoding="UTF-16")
```

text
The text content. A unicode string that specifies the TEXT record text field. Coerced into unicode when set.

language
The text language. A string that specifies the ISO/IANA language code coded into the TEXT record. The value is not verified except that a ValueError exception is raised if the assigned value string exceeds 64 characters.

encoding
The text encoding, given as a string. May be 'UTF-8' or 'UTF-16'. A ValueError exception is raised for anythinge else.

8.3.4 nfc.ndef.UriRecord

class nfc.ndef.**UriRecord**(*uri=None*)
Bases: nfc.ndef.record.Record

Wraps an NDEF URI record and provides access to the *uri* content. The URI RTD specification defines the payload of the URI record as a URI identifier code byte followed by a URI string. The URI identifier code provides one byte code points for abbreviations of commonly used URI protocol names. The *UriRecord* class handles abbreviations transparently by expanding and compressing when decoding and encoding.

Parameters **uri** – URI string or *nfc.ndef.Record* object

The *uri* argument may alternatively supply an instance of class *nfc.ndef.Record*. Initialization is then done by parsing the record payload. If the record type does not match 'urn:nfc:wkt:U' a ValueError exception is raised.

```
>>> nfc.ndef.UriRecord(nfc.ndef.Record())
>>> nfc.ndef.UriRecord("http://nfcpy.org")
```

uri
The URI string, including any abbreviation that is possibly available. A ValueError exception is raised if the string contains non ascii characters.

8.3.5 nfc.ndef.SmartPosterRecord

class nfc.ndef.**SmartPosterRecord**(*uri*, *title={}*, *icons={}*, *action='default'*, *resource_size=None*, *resource_type=None*)

Bases: nfc.ndef.record.Record

Wraps an NDEF SmartPoster record and provides access to the `encoding`, `language` and actual `text` content.

Parameters

- **uri** – URI string or *nfc.ndef.Record* object
- **title** – Smart poster title(s), assigned to *title*
- **icons** – Smart poster icons, assigned to *icons*
- **action** – Recommended action, assigned to *action*
- **resource_size** – Size of the referenced resource
- **resource_type** – Type of the referenced resource

The *uri* argument may alternatively supply an instance of class *nfc.ndef.Record*. Initialization is then done by parsing the record payload. If the record type does not match 'urn:nfc:wkt:Sp' a `ValueError` exception is raised.

```
>>> nfc.ndef.SmartPosterRecord(nfc.ndef.Record())
>>> nfc.ndef.SmartPosterRecord("http://nfcpy.org", "nfcpy")
>>> nfc.ndef.SmartPosterRecord("http://nfcpy.org", "nfcpy", action="save")
```

uri
The smart poster URI, a string of ascii characters. A `ValueError` exception is raised if non ascii characters are contained.

title
A dictionary of smart poster titles with ISO/IANA language codes as keys and title strings as values. Set specific title strings with `obj.title['en']=title`. Assigning a string value is equivalent to setting the title for language code 'en'. Titles are optional for a smart poster record

icons
A dictionary of smart poster icon images. The keys specify the image mime sub-type and the values are strings of image data. Icons are optional for a smart poster record.

action
The recommended action for the receiver of the smart poster. Reads as 'default', 'exec', 'save', 'edit' or a number string if RFU values were decoded. Can be set to 'exec', 'save', 'edit' or `None`. The action is optional in a smart poster record.

resource_size
The size of the resource referred by the URI. A 32 bit unsigned integer value or `None`. The resource size is optional in a smart poster record.

resource_type
The type of the resource referred by the URI. A UTF-8 formatted string that describes an Internet media type (MIME type) or `None`. The resource type is optional in a smart poster record.

8.3.6 nfc.ndef.HandoverRequestMessage

class nfc.ndef.**HandoverRequestMessage**(*message=None*, *version=None*)

The handover request message is used in the the NFC Connection Handover protocol to send proposals for

alternative carriers to a peer device.

Parameters

- **message** (*nfc.ndef.Message*) – a parsed message with type 'urn:nfc:wkt:Hr'
- **version** (str) – a '<major-number>.<minor-number>' version string

Either the *message* or *version* argument must be supplied. A ValueError is raised if both arguments are present or absent.

The *message* argument must be a parsed NDEF message with, according to the Connection Handover Specification, at least two records. The first record, and thus the message, must match the NFC Forum Well-Known Type 'urn:nfc:wkt:Hr'.

The version argument indicates the Connection Handover version that shall be used for encoding the handover request message NDEF data. It is currently limited to major-version '1' and minor-version '0' to '15' and for any other value a ValueError exception is raised.

```
>>> nfc.ndef.HandoverRequestMessage(nfc.ndef.Message(ndef_message_data))
>>> nfc.ndef.HandoverRequestMessage(version='1.2')
```

type
The message type. This is a read-only attribute which returns the NFC Forum Well-Known Type 'urn:nfc:wkt:Hr'

name
The message name (identifier). Corresponds to the name of the handover request record.

version
Connection Handover version number that the messsage complies to. A read-only *Version* object that provides the major and minor version int values.

nonce
A nonce received or to be send as the random number for handover request collision resolution. This attribute is supported only since version 1.2.

carriers
List of alternative carriers. Each entry is an *Carrier* object that holds properties of the alternative carrier. Use *add_carrier()* to expand this list.

add_carrier(*carrier_record*, *power_state*, *aux_data_records=None*)
Add a new carrier to the handover request message.

Parameters

- **carrier_record** (*nfc.ndef.Record*) – a record providing carrier information
- **power_state** (str) – a string describing the carrier power state
- **aux_data_records** (*RecordList*) – list of auxiliary data records

```
>>> hr = nfc.ndef.HandoverRequestMessage(version="1.2")
>>> hr.add_carrier(some_carrier_record, "active")
```

pretty(*indent=0*)
Returns a string with a formatted representation that might be considered pretty-printable.

8.3.7 nfc.ndef.HandoverSelectMessage

class nfc.ndef.**HandoverSelectMessage**(*message=None*, *version=None*)
The handover select message is used in the the NFC Connection Handover protocol to send agreements for

alternative carriers to a peer device as response to a handover request message.

Parameters

- **message** (*nfc.ndef.Message*) – a parsed message with type 'urn:nfc:wkt:Hs'
- **version** (str) – a '<major-number>.<minor-number>' version string

Either the *message* or *version* argument must be supplied. A ValueError is raised if both arguments are present or absent.

The *message* argument must be a parsed NDEF message with, according to the Connection Handover Specification, at least one record. The first record, and thus the message, must match the NFC Forum Well-Known Type 'urn:nfc:wkt:Hs'.

The version argument indicates the Connection Handover version that shall be used for encoding the handover select message NDEF data. It is currently limited to major-version '1' and minor-version '0' to '15' and for any other value a ValueError exception is raised.

```
>>> nfc.ndef.HandoverSelectMessage(nfc.ndef.Message(ndef_message_data))
>>> nfc.ndef.HandoverSelectMessage(version='1.2')
```

type
The message type. This is a read-only attribute which returns the NFC Forum Well-Known Type 'urn:nfc:wkt:Hs'

name
The message name (identifier). Corresponds to the name of the handover select record.

version
Connection Handover version number that the messsage complies to. A read-only *Version* object that provides the major and minor version int values.

error
A *HandoverError* structure that provides error reason and data received or to be send with the handover select message. An error.reason value of 0 means that no error was received or is to be send.

carriers
List of alternative carriers. Each entry is an *Carrier* object that holds properties of the alternative carrier. Use *add_carrier()* to expand this list.

add_carrier(*carrier_record*, *power_state*, *aux_data_records=[]*)
Add a new carrier to the handover select message.

Parameters

- **carrier_record** (*nfc.ndef.Record*) – a record providing carrier information
- **power_state** (str) – a string describing the carrier power state
- **aux_data_records** (*RecordList*) – list of auxiliary data records

```
>>> hs = nfc.ndef.HandoverSelectMessage(version="1.2")
>>> hs.add_carrier(some_carrier_record, "active")
```

pretty(*indent=0*)
Returns a string with a formatted representation that might be considered pretty-printable.

8.3.8 nfc.ndef.HandoverCarrierRecord

class nfc.ndef.**HandoverCarrierRecord**(*carrier_type*, *carrier_data=None*)
Bases: nfc.ndef.record.Record

The handover carrier record is used to identify an alternative carrier technology in a handover request message when no carrier configuration data shall be transmitted.

Parameters

- **carrier_type** (str) – identification of an alternative carrier
- **carrier_data** (str) – additional alternative carrier information

```
>>> nfc.ndef.HandoverCarrierRecord('application/vnd.bluetooth.ep.oob')
```

carrier_type
Identification of an alternative carrier. A string formatted as an NFC Forum Well-Known or External Type or Internet Media Type or absolute URI. This attribute is read-only.

carrier_data
An octet string that provides additional information about the alternative carrier.

8.3.9 nfc.ndef.handover.Version

class nfc.ndef.handover.**Version**

major
Major version number. A read-only attribute.

minor
Mainor version number. A read-only attribute.

8.3.10 nfc.ndef.handover.Carrier

class nfc.ndef.handover.**Carrier**

type
The alternative carrier type name, equivalent to `Carrier.record.type` or `Carrier.record.carrier_type` if the carrier is specified as a *HandoverCarrierRecord*.

record
A carrier configuration record. Recognized and further interpreted records are: *HandoverCarrierRecord*, *BluetoothConfigRecord*, *WifiConfigRecord*, *WifiPasswordRecord*.

power_state
The carrier power state. This may be one of the following strings: “inactive”, “active”, “activating”, or “unknown”.

auxiliary_data_records
A list of auxiliary data records providing additional carrier information.

8.3.11 nfc.ndef.handover.HandoverError

class nfc.ndef.handover.**HandoverError**

reason
The error reason. An 8-bit unsigned integer.

data
The error data. An 8-bit unsigned integer if `reason` is 1 or 3, a 32-bit unsigned integer if `reason` is 2.

8.3.12 nfc.ndef.BluetoothConfigRecord

class `nfc.ndef.`**`BluetoothConfigRecord`**
Bases: `nfc.ndef.record.Record`

device_address
Bluetooth device address. A string of hexadecimal characters with 8-bit quantities spearated by colons and the most significant byte first. For example, the device address `'01:23:45:67:89:AB'` corresponds to `0x0123456789AB`.

local_device_name
Bluetooth Local Name encoded as sequence of characters in the given order. Received as complete (EIR type 0x09) or shortened (EIR type 0x08) local name. Transmitted as complete local name. Set to None if not received or not to be transmitted.

simple_pairing_hash
Simple Pairing Hash C. Received and transmitted as EIR type 0x0E. Set to None if not received or not to be transmitted. Raises nfc.ndef.DecodeError if the received value or nfc.ndef.EncodeError if the assigned value is not a sequence of 16 octets.

simple_pairing_rand
Simple Pairing Randomizer R. Received and transmitted as EIR type 0x0F. Set to None if not received or not to be transmitted. Raises nfc.ndef.DecodeError if the received value or nfc.ndef.EncodeError if the assigned value is not a sequence of 16 octets.

service_class_uuid_list
Listq of Service Class UUIDs. Set and retrieved as a list of complete 128-bit UUIDs. Decoded from and encoded as EIR types 0x02/0x03 (16-bit partial/complete UUIDs), 0x04/0x05 (32-bit partial/complete UUIDs), 0x06/0x07 (128-bit partial/complete UUIDs).

class_of_device
Class of Device encoded as unsigned long integer. Received and transmitted as EIR type 0x0D in little endian byte order. Set to None if not received or not to be transmitted.

8.3.13 nfc.ndef.WifiConfigRecord

class `nfc.ndef.`**`WifiConfigRecord`**
Bases: `nfc.ndef.record.Record`

version
The WiFi Simple Configuration version, coded as a 'major.minor' string

credentials
A list of WiFi credentials. Each credential is a dictionary with any of the possible keys `'network-name'`, `'network-key'`, `'shareable'`, `'authentication'`, `'encryption'`, `'mac-address'`, and `'other'`.

credential
The first WiFi credential. Same as `WifiConfigRecord().credentials[0]`.

other
A list of WiFi attribute (key, value) pairs other than version and credential(s). Keys are two character strings for standard WiFi attributes, one character strings for subelements within a WFA vendor extension attribute, and three character strings for other vendor ecxtension attributes.

8.3.14 nfc.ndef.WifiPasswordRecord

class nfc.ndef.**WifiPasswordRecord**
Bases: nfc.ndef.record.Record

version
The WiFi Simple Configuration version, coded as a 'major.minor' string

passwords
A list of WiFi out-of-band device passwords. Each password is a dictionary with the keys 'public-key-hash', 'password-id', and 'password'.

password
The first WiFi device password. Same as WifiPasswordRecord().passwords[0].

other
A list of WiFi attribute (key, value) pairs other than version and device password. Keys are two character strings for standard WiFi attributes, one character strings for subelements within a WFA vendor extension attribute, and three character strings for other vendor extension attributes.

8.4 nfc.llcp

The nfc.llcp module implements the NFC Forum Logical Link Control Protocol (LLCP) specification and provides a socket interface to use the connection-less and connection-mode transport facilities of LLCP.

8.4.1 nfc.llcp.Socket

class nfc.llcp.**Socket**(*llc*, *sock_type*)
Create a new LLCP socket with the given socket type. The socket type should be one of:

- nfc.llcp.LOGICAL_DATA_LINK for best-effort communication using LLCP connection-less PDU exchange
- nfc.llcp.DATA_LINK_CONNECTION for reliable communication using LLCP connection-mode PDU exchange
- nfc.llcp.llc.RAW_ACCESS_POINT for unregulated LLCP PDU exchange (useful to implement test programs)

llc
The LogicalLinkController instance to which this socket belongs. This attribute is read-only.

resolve(*name*)
Resolve a service name into an address. This may involve conversation with the remote service discovery component if the name is hasn't yet been resolved. The return value is the service access point address that the service name is bound to at the remote device. A zero address indicates that the remote device does not know about the service name requested. The return value is None if communication with the peer device got terminated.

setsockopt(*option*, *value*)
Set the value of the given socket option and return the current value which may have been corrected if it was out of bounds.

getsockopt(*option*)
Return the value of the given socket option.

bind(*address=None*)
Bind the socket to address. The socket must not already be bound. The address may be a service name string, a service access point number, or it may be omitted. If address is a well-known service name the socket will be bound to the corresponding service access point address, otherwise the socket will be bound to the next available service access point address between 16 and 31 (inclusively). If address is a number between 32 and 63 (inclusively) the socket will be bound to that service access point address. If the address argument is omitted the socket will be bound to the next available service access point address between 32 and 63.

connect(*address*)
Connect to a remote socket at address. Address may be a service name string or a service access point number.

listen(*backlog*)
Mark a socket as a socket that will be used to accept incoming connection requests using accept(). The *backlog* defines the maximum length to which the queue of pending connections for the socket may grow. A backlog of zero disables queuing of connection requests.

accept()
Accept a connection. The socket must be bound to an address and listening for connections. The return value is a new socket object usable to send and receive data on the connection.

send(*string*)
Send data to the socket. The socket must be connected to a remote socket. Returns a boolean value that indicates success or failure. Failure to send is generally an indication that the socket or connection was closed.

sendto(*string*, *address*)
Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by address. Returns a boolean value that indicates success or failure. Failure to send is generally an indication that the socket was closed.

recv()
Receive data from the socket. The return value is a string representing the data received. The maximum amount of data that may be returned is determined by the link or connection maximum information unit size.

recvfrom()
Receive data from the socket. The return value is a pair (string, address) where string is a string representing the data received and address is the address of the socket sending the data.

poll(*event*, *timeout=None*)
Wait for a socket event.

getsockname()
Obtain the address to which the socket is bound. For an unbound socket the returned value is None.

getpeername()
Obtain the address of the peer connected on the socket. For an unconnected socket the returned value is None.

close()
Close the socket. All future operations on the socket object will fail. The remote end will receive no more data Sockets are automatically closed when the logical link controller terminates (gracefully or by link disruption). A connection-mode socket will attempt to disconnect the data link connection (if in connected state).

8.4.2 nfc.llcp.llc.LogicalLinkController

class nfc.llcp.llc.**LogicalLinkController**(*recv_miu=248*, *send_lto=500*, *send_agf=True*, *symm_log=True*)

8.5 nfc.snep

The nfc.snep module implements the NFC Forum Simple NDEF Exchange Protocol (SNEP) specification and provides a server and client class for applications to easily send or receive SNEP messages.

8.5.1 nfc.snep.SnepServer

class nfc.snep.**SnepServer**(*llc*, *service_name='urn:nfc:sn:snep'*, *max_acceptable_length=1048576*, *recv_miu=1984*, *recv_buf=15*)

NFC Forum Simple NDEF Exchange Protocol server

get(*acceptable_length*, *ndef_message*)

Handle Get requests. This method should be overwritten by a subclass of SnepServer to customize it's behavior. The default implementation simply returns Not Implemented.

put(*ndef_message*)

Handle Put requests. This method should be overwritten by a subclass of SnepServer to customize it's behavior. The default implementation simply returns Not Implemented.

8.5.2 nfc.snep.SnepClient

class nfc.snep.**SnepClient**(*llc*, *max_ndef_msg_recv_size=1024*)

Simple NDEF exchange protocol - client implementation

connect(*service_name*)

Connect to a SNEP server. This needs only be called to connect to a server other than the Default SNEP Server at *urn:nfc:sn:snep* or if the client wants to send multiple requests with a single connection.

close()

Close the data link connection with the SNEP server.

get(*ndef_message=None*, *timeout=1.0*)

Get an NDEF message from the server. Temporarily connects to the default SNEP server if the client is not yet connected.

put(*ndef_message*, *timeout=1.0*)

Send an NDEF message to the server. Temporarily connects to the default SNEP server if the client is not yet connected.

8.6 nfc.handover

The nfc.handover module implements the NFC Forum Connection Handover 1.2 protocol as a server and client class that simplify realization of handover selector and requester functionality.

8.6.1 nfc.handover.HandoverServer

class nfc.handover.**HandoverServer**(*llc*, *request_size_limit=65536*, *recv_miu=1984*, *recv_buf=15*)
NFC Forum Connection Handover server

process_request(*request*)
Process a handover request message. The *request* argument is a *nfc.ndef.HandoverRequestMessage* object. The return value must be a *nfc.ndef.HandoverSelectMessage* object to be sent back to the client.

This method should be overwritten by a subclass of *HandoverServer* to customize it's behavior. The default implementation returns a version 1.2 *nfc.ndef.HandoverSelectMessage* with no carriers.

8.6.2 nfc.handover.HandoverClient

class nfc.handover.**HandoverClient**(*llc*)
NFC Forum Connection Handover client

connect(*recv_miu=248*, *recv_buf=2*)
Connect to the remote handover server if available. Raises nfc.llcp.ConnectRefused if the remote device does not have a handover service or the service does not accept any more connections.

close()
Disconnect from the remote handover server.

send(*message*)
Send a handover request message to the remote server.

recv(*timeout=None*)
Receive a handover select message from the remote server.

n

Symbols

A

B

C

M

N

O

P

R

S

T

U

V

W