# ndeflib documentation

***Release 0.3.3***

**Stephen Tiedemann**

**Sep 07, 2020**

# Documentation

The ndeflib is a Python package for parsing and generating NFC Data Exchange Format (NDEF) messages. It is licensed under the ISCL, hosted on GitHub and can be installed from PyPI.

```
>>> import ndef
>>> hexstr = '9101085402656e48656c6c6f5101085402656e576f726c64'
>>> octets = bytearray.fromhex(hexstr)
>>> for record in ndef.message_decoder(octets): print(record)
NDEF Text Record ID '' Text 'Hello' Language 'en' Encoding 'UTF-8'
NDEF Text Record ID '' Text 'World' Language 'en' Encoding 'UTF-8'
>>> message = [ndef.TextRecord("Hello"), ndef.TextRecord("World")]
>>> b''.join(ndef.message_encoder(message)) == octets
True
```

CHAPTER 1

# NDEF Decoding and Encoding

NDEF (NFC Data Exchange Format), specified by the NFC Forum, is a binary message format used to encapsulate application-defined payloads exchanged between NFC Devices and Tags. Each payload is encoded as an NDEF Record with fields that specify the payload size, payload type, an optional payload identifier, and flags for indicating the first and last record of an NDEF Message or tagging record chunks. An NDEF Message is simply a sequence of one or more NDEF Records where the first and last record are marked by the Message Begin and End flags.

The `ndef` package interface for decoding and encoding of NDEF Messages consists of the *`message_decoder()`* and *`message_encoder()`* functions that both return generators for decoding octets into `ndef.Record` instances or encoding `ndef.Record` instances into octets. *Known record types* are decoded into instances of their implementation class and can be directly encoded as part of a message.

## 1.1 Message Decoder

**`message_decoder`**(*stream_or_bytes*, *errors='strict'*, *known_types=Record._known_types*)

Returns a generator function that decodes NDEF Records from a file-like, byte-oriented stream or a bytes object given by the *stream_or_bytes* argument. When the *errors* argument is set to 'strict' (the default), the decoder expects a valid NDEF Message with Message Begin and End flags set for the first and last record and decoding of known record types will fail for any format errors. Minor format errors are accepted when *errors* is set to 'relax'. With *errors* set to 'ignore' the decoder silently stops when a non-correctable error is encountered. The *known_types* argument provides the mapping of record type strings to class implementations. It defaults to all global records implemented by `ndeflib` or additionally registered from user code. It's main use would probably be to force decoding into only generic records with `known_types={}`.

**Parameters**

- **`stream_or_bytes`** (*`byte stream or bytes object`*) – message data octets
- **`errors`** (*`str`*) – error handling strategy, may be 'strict', 'relax' or 'ignore'
- **`known_types`** (*`dict`*) – mapping of known record types to implementation classes

**Raises** **`ndef.DecodeError`** – for data format errors (unless *errors* is set to 'ignore')

```
>>> import ndef
>>> octets = bytearray.fromhex('910303414243616263 5903030144454630646566')
>>> decoder = ndef.message_decoder(octets)
>>> next(decoder)
ndef.record.Record('urn:nfc:wkt:ABC', '', bytearray(b'abc'))
>>> next(decoder)
ndef.record.Record('urn:nfc:wkt:DEF', '0', bytearray(b'def'))
>>> next(decoder)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> message = list(ndef.message_decoder(octets))
>>> len(message)
2
```

## 1.2 Message Encoder

**message_encoder**(*message=None*, *stream=None*)

Returns a generator function that encodes `ndef.Record` objects into an NDEF Message octet sequence. The *message* argument is either an iterable of records or None, if *message* is None the records must be sequentially send to the encoder (as for any generator the first send value must be None, specific to the message encoder is that octets are generated for the previous record and a final None value must be send for the last record octets). The *stream* argument controls the output of the generator function. If *stream* is None, the generator yields a bytes object for each encoded record. Otherwise, it must be a file-like, byte-oriented stream that receives the encoded octets and the generator yields the number of octets written per record.

**Parameters**

- **message** (*iterable or None*) – sequence of records to encode
- **stream** (*byte stream or None*) – file-like output stream

**Raises** **ndef.EncodeError** – for invalid record parameter values or types

```
>>> import ndef
>>> record1 = ndef.Record('urn:nfc:wkt:ABC', '1', b'abc')
>>> record2 = ndef.Record('urn:nfc:wkt:DEF', '2', b'def')
>>> encoder = ndef.message_encoder()
>>> encoder.send(None)
>>> encoder.send(record1)
>>> encoder.send(record2)
b'\x99\x03\x03\x01ABC1abc'
>>> encoder.send(None)
b'Y\x03\x03\x01DEF2def'
>>> message = [record1, record2]
>>> b''.join((ndef.message_encoder(message)))
b'\x99\x03\x03\x01ABC1abcY\x03\x03\x01DEF2def'
>>> list((ndef.message_encoder(message, open('/dev/null', 'wb'))))
[11, 11]
```

## 1.3 Record Class

**class Record**(*type=''*, *name=''*, *data=b''*)

This class implements generic decoding and encoding of an NDEF Record and is the base for all specialized

record type classes. The NDEF Record Payload Type encoded by the TNF (Type Name Format) and TYPE field is represented by a single *type* string argument:

*Empty (TNF 0)*

An *Empty* record has no TYPE, ID, and PAYLOAD fields. This is set if the *type* argument is absent, None, or an empty string. Encoding ignores whatever is set as *name* and *data*, producing just the short length record `b'\x10\x00\x00'`.

*NFC Forum Well Known Type (TNF 1)*

An *NFC Forum Well Known Type* is a URN (**RFC 2141**) with namespace identifier (NID) `nfc` and the namespace specific string (NSS) prefixed with `wkt:`. When encoding, the type is written as a relative-URI (cf. **RFC 3986**), omitting the NID and the prefix. For example, the type `urn:nfc:wkt:T` is encoded as TNF 1, TYPE `T`.

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

A *media-type* follows the media-type grammar defined in **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 payloads 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 `http://schemas.xmlsoap.org/soap/envelope/`.

*NFC Forum External Type (TNF 4)*

An *NFC Forum External Type* is a URN (**RFC 2141**) with namespace identifier (NID) `nfc` and the namespace specific string (NSS) prefixed with `ext:`. When encoding, the type is written as a relative-URI (cf. **RFC 3986**), omitting the NID and the prefix. For example, the type `urn:nfc:ext:nfcpy.org:T` will be encoded as TNF 4, TYPE `nfcpy.org:T`.

*Unknown (TNF 5)*

The *Unknown* record type indicates that the type of the payload is unknown, similar to the `application/octet-stream` media type. It is set with the *type* argument `unknown` and encoded with an empty TYPE field.

*Unchanged (TNF 6)*

The *Unchanged* record type is used for all except the first record in a chunked payload. It is set with the *type* argument `unchanged` and encoded with an empty TYPE field.

The *type* argument sets the final value of the *type* attribute, which provides the value only for reading. The *name* and *data* argument set the initial values of the *name* and *data* attributes. They can both be changed later.

**Parameters**

- **type** (*str*) – final value for the *type* attribute
- **name** (*str*) – initial value for the see *name* attribute
- **data** (*bytes*) – initial value for the *data* attribute

**type**
The record type is a read-only text string set either by decoding or through initialization.

**name**
The record name is a text string that corresponds to the NDEF Record ID field. The maximum capacity is 255 8-bit characters, converted in and out as latin-1.

**data**
The record data is a bytearray with the sequence of octets that correspond to the NDEF Record PAYLOAD field. The attribute itself is readonly but the bytearray content can be changed. Note that for derived record classes this becomes a read-only bytes object with the content encoded from the record's attributes.

**MAX_PAYLOAD_SIZE**
This is a class data attribute that restricts the decodable and encodable maximum NDEF Record PAYLOAD size from the theoretical value of up to 4GB to 1MB. If needed, a different value can be assigned to the record class: `ndef.Record.MAX_PAYLOAD_SIZE = 100*1024`

**classmethod register_type**(*record_class*)
Register a derived record class as a known type for decoding. This creates an entry for the record_class type string to be decoded as a record_class instance. Beyond internal use this is needed for *adding private records*.

# CHAPTER 2

# Known Record Types

The ndef package implements special decoding and encoding for a number of known record types.

## 2.1 Text Record

The NDEF Text Record is a well-known record type defined by the NFC Forum. It carries a UTF-8 or UTF-16 encoded text string with an associated IANA language code identifier.

**class TextRecord**(*text=''*, *language='en'*, *encoding='UTF-8'*)

A *TextRecord* is initialized with the actual text content, an ISO/IANA language identifier, and the desired transfer encoding UTF-8 or UTF-16. Default values are empty text, language code 'en', and 'UTF-8' encoding.

**Parameters**

- **text** (*str*) – initial value for the *text* attribute, default ''
- **language** (*str*) – initial value for the *language* attribute, default 'en'
- **encoding** (*str*) – initial value for the *encoding* attribute, default 'UTF-8'

**type**

The Text Record type is urn:nfc:wkt:T.

**name**

Value of the NDEF Record ID field, an empty str if not set.

**data**

A bytes object containing the NDEF Record PAYLOAD encoded from the current attributes.

**text**

The decoded or set text string value.

**language**

The decoded or set IANA language code identifier.

**encoding**

The transfer encoding of the text string. Either 'UTF-8' or 'UTF-16'.

```
>>> import ndef
>>> record = ndef.TextRecord("Hallo Welt", "de")
>>> octets = b''.join(ndef.message_encoder([record]))
>>> print(list(ndef.message_decoder(octets))[0])
NDEF Text Record ID '' Text 'Hallo Welt' Language 'de' Encoding 'UTF-8'
```

## 2.2 URI Record

The NDEF URI Record is a well-known record type defined by the NFC Forum. It carries a, potentially abbreviated, UTF-8 encoded Internationalized Resource Identifier (IRI) as defined by **RFC 3987**. Abbreviation covers certain prefix patterns that are compactly encoded as a single octet and automatically expanded when decoding. The *UriRecord* class provides both access attributes for decoded IRI as well as a converted URI (if a netloc part is present in the IRI).

**class UriRecord**(*iri=''*)

The *UriRecord* class decodes or encodes an NDEF URI Record. The *UriRecord.iri* attribute holds the expanded (if a valid abbreviation code was decoded) internationalized resource identifier (IRI). The *UriRecord.uri* attribute is a converted version of the IRI. Conversion is applied only for IRI's that split with a netloc component. A converted URI contains only ASCII characters with an IDNA encoded netloc component and percent-encoded path, query and fragment components.

**Parameters** **iri** (*str*) – initial value for the *iri* attribute, default ''

**type**

The URI Record type is `urn:nfc:wkt:U`.

**name**

Value of the NDEF Record ID field, an empty `str` if not set.

**data**

A `bytes` object containing the NDEF Record PAYLOAD encoded from the current attributes.

**iri**

The decoded or set internationalized resource identifier, expanded if an abbreviation code was used in the record payload.

**uri**

The uniform resource identifier translated from the *UriRecord.iri* attribute.

```
>>> import ndef
>>> record = ndef.UriRecord("http://www.hääyö.com/~user/")
>>> record.iri
'http://www.hääyö.com/~user/'
>>> record.uri
'http://www.xn--hy-viaa5g.com/~user/'
>>> record = ndef.UriRecord("http://www.example.com")
>>> b''.join(ndef.message_encoder([record]))
b'\xd1\x01\x0cU\x01example.com'
```

## 2.3 Smartposter Record

The NFC Forum Smart Poster Record Type Definition defines a structure that associates an Internationalized Resource Identifier (or Uniform Resource Identifier) with various types of metadata. For a user this is most noteably the ability to attach descriptive text in different languages as well as image data for icon rendering. For a smartposter application

this is a recommendation for processing as well as resource type and size hints to guide a strategy for retrieving the resource.

**class SmartposterRecord**(*resource*, *title=None*, *action=None*, *icon=None*, *resource_size=None*, *resource_type=None*)

Initialize a *SmartposterRecord* instance. The only required argument is the Internationalized Resource Identifier *resource*, all other arguments are optional metadata.

**Parameters**

- **resource** (*str*) – Internationalized Resource Identifier
- **title** (*str or dict*) – English title `str` or `dict` with language keys and title values
- **action** (*str or int*) – assigns a value to the *action* attribute
- **icon** (*bytes or dict*) – PNG data `bytes` or `dict` with {icon-type: icon_data} items
- **resource_size** (*int*) – assigns a value to the *resource_size* attribute
- **resource_type** (*str*) – assigns a value to the *resource_type* attribute

**type**
The Smartposter Record type is `urn:nfc:wkt:Sp`.

**name**
Value of the NDEF Record ID field, an empty `str` if not set.

**data**
A `bytes` object containing the NDEF Record PAYLOAD encoded from the current attributes.

**resource**
Get or set the Smartposter resource identifier. A set value is interpreted as an internationalized resource identifier (so it can be unicode). When reading, the resource attribute returns a *UriRecord* which can be used to set the *UriRecord.iri* and *UriRecord.uri* directly.

**title**
The title string for language code 'en' or the first title string that was decoded or set. If no title string is available the value is `None`. The attribute can not be set, use *set_title()*.

**titles**
A dictionary of all decoded or set titles with language `str` keys and title `str` values. The attribute can not be set, use *set_title()*.

**set_title**(*title*, *language='en'*, *encoding='UTF-8'*)
Set the title string for a specific language which defaults to 'en'. The transfer encoding may be set to either 'UTF-8' or 'UTF-16', the default is 'UTF-8'.

**action**
Get or set the recommended action for handling the Smartposter resource. A set value may be 'exec', 'save', 'edit' or an index thereof. A read value is either one of above strings or `None` if no action value was decoded or set.

**icon**
The image data `bytes` for an 'image/png' type smartposter icon or the first icon decoded or added. If no icon is available the value is `None`. The attribute can not be set, use *add_icon()*.

**icons**
A dictionary of icon images with mime-type `str` keys and icon-data `bytes` values. The attribute can not be set, use *add_icon()*.

**add_icon**(*icon_type*, *icon_data*)
Add a Smartposter icon as icon_data bytes for the image or video mime-type string supplied with icon_type.

**resource_size**
Get or set the `int` size hint for the Smartposter resource. `None` if a size hint was not decoded or set.

**resource_type**
Get or set the `str` type hint for the Smartposter resource. `None` if a type hint was not decoded or set.

```
>>> import ndef
>>> record = ndef.SmartposterRecord('https://github.com/nfcpy/ndeflib')
>>> record.set_title('Python package for parsing and generating NDEF', 'en')
>>> record.resource_type = 'text/html'
>>> record.resource_size = 1193970
>>> record.action = 'exec'
>>> len(b''.join(ndef.message_encoder([record])))
115
```

## 2.4 Device Information Record

The NDEF Device Information Record is a well-known record type defined by the NFC Forum. It carries a number of Type-Length-Value data elements that provide information about the device, such as the manufacturer and device model name.

**class DeviceInformationRecord**(*vendor_name*, *model_name*, *unique_name=None*, *uuid_string=None*, *version_string=None*)
Initialize the record with required and optional device information. The vendor_name and model_name arguments are required, all other arguments are optional information.

**Parameters**

- **vendor_name** (*str*) – sets the *vendor_name* attribute
- **model_name** (*str*) – sets the *model_name* attribute
- **unique_name** (*str*) – sets the *unique_name* attribute
- **uuid_string** (*str*) – sets the *uuid_string* attribute
- **version_string** (*str*) – sets the *version_string* attribute

**type**
The Device Information Record type is `urn:nfc:wkt:Di`.

**name**
Value of the NDEF Record ID field, an empty `str` if not set.

**data**
A `bytes` object containing the NDEF Record PAYLOAD encoded from the current attributes.

**vendor_name**
Get or set the device vendor name `str`.

**model_name**
Get or set the device model name `str`.

**unique_name**
Get or set the device unique name `str`.

**uuid_string**
Get or set the universially unique identifier `str`.

**version_string**
Get or set the device firmware version `str`.

**undefined_data_elements**
A list of undefined data elements as named tuples with data_type and data_bytes attributes. This is a reference to the internal list and may thus be updated in-place but it is strongly recommended to use the add_undefined_data_element method with data_type and data_bytes validation. It would also not be safe to rely on such implementation detail.

**add_undefined_data_element**(*data_type*, *data_bytes*)
Add an undefined (reserved future use) device information data element. The data_type must be an an integer in range(5, 256). The data_bytes argument provides the up to 255 octets to transmit.

Undefined data elements should not normally be added. This method is primarily here to allow data elements defined by future revisions of the specification before this implementation is updated.

```
>>> import ndef
>>> record = ndef.DeviceInformationRecord('Sony', 'RC-S380')
>>> record.unique_name = 'Black NFC Reader connected to PC'
>>> record.uuid_string = '123e4567-e89b-12d3-a456-426655440000'
>>> record.version_string = 'NFC Port-100 v1.02'
>>> len(b''.join(ndef.message_encoder([record])))
92
```

## 2.5 Connection Handover

The NFC Forum Connection Handover specification defines a number of Record structures that are used to exchange messages between Handover Requester, Selector and Mediator devices to eventually establish alternative carrier connections for additional data exchange. Generally, a requester device sends a Handover Request Message to announce supported alternative carriers and expects the selector device to return a Handover Select Message with a selection of alternative carriers supported by both devices. If the two devices are not close enough for NFC communication, a third device may use the Handover Mediation and Handover Initiate Messages to relay information between the two.

Any of above mentioned Handover Messages is constructed as an NDEF Message where the first record associates the processing context. The Handover Request, Select, Mediation, and Initiate Record classes implement the appropriate context, i.e. record types known by context are decoded by associated record type classes while others are decoded as generic NDEF Records.

### 2.5.1 Handover Request Record

The Handover Request Record is the first record of a connection handover request message. Information enclosed within the payload of a handover request record includes the handover version number, a random number for resolving a handover request collision (when both peer devices simultaenously send a handover request message) and a number of references to alternative carrier information records subsequently encoded in the same message.

```
>>> import ndef
>>> from os import urandom
>>> wsc = 'application/vnd.wfa.wsc'
>>> message = [ndef.HandoverRequestRecord('1.3', urandom(2))]
>>> message.append(ndef.HandoverCarrierRecord(wsc, None, 'wifi'))
>>> message[0].add_alternative_carrier('active', message[1].name)
```

**class HandoverRequestRecord**(*version='1.3'*, *crn=None*, **alternative_carrier*)
Initialize the record with a *version* number, a collision resolution random number *crn* and zero or more *alternative_carrier*. The version number can be set as an 8-bit integer (with 4-bit major and minor part), or as a `'{major}.{minor}'` version string. An alternative carrier is given by a tuple with *carrier power state*,

*carrier data reference* and zero or more *auxiliary data references*. The collision resolution number (crn) argument is the unsigned 16-bit random integer for connection handover version '1.2' or later, for any prior version number it must be None.

**Parameters**

- **version** (*int or str*) – handover version number
- **crn** (*int*) – collision resolution random number
- **alternative_carrier** (*tuple*) – alternative carrier entry

**type**
The Handover Request Record type is `urn:nfc:wkt:Hr`.

**name**
Value of the NDEF Record ID field, an empty `str` if not set.

**data**
A `bytes` object containing the NDEF Record PAYLOAD encoded from the current attributes.

**hexversion**
The version as an 8-bit integer with 4-bit major and minor part. This is a read-only attribute.

**version_info**
The version as a named tuple with major and minor version number attributes. This is a read-only attribute.

**version_string**
The version as the '{major}.{minor}' formatted string. This is a read-only attribute.

**collision_resolution_number**
Get or set the random number for handover request message collision resolution. May be None if the random number was neither decoded or set.

**alternative_carriers**
A `list` of alternative carriers with attributes carrier_power_state, carrier_data_reference, and auxiliary_data_reference list.

**add_alternative_carrier(cps, cdr, *adr):**
Add a reference to a carrier data record within the handover request message. The carrier data reference *cdr* is the name (NDEF Record ID) of the carrier data record. The carrier power state *cps* is either 'inactive', 'active', 'activating', or 'unknown'. Any number of auxiliary data references *adr* may be added to link with other records in the message that carry information related to the carrier.

### 2.5.2 Handover Select Record

The Handover Select Record is the first record of a connection handover select message. Information enclosed within the payload of a handover select record includes the handover version number, error reason and associated error data when processing of the previously received handover request message failed, and a number of references to alternative carrier information records subsequently encoded in the same message.

```
>>> import ndef
>>> carrier = ndef.Record('mimetype/subtype', 'ref', b'1234')
>>> message = [ndef.HandoverSelectRecord('1.3'), carrier]
>>> message[0].add_alternative_carrier('active', carrier.name)
```

**class HandoverSelectRecord**(*version='1.3'*, *error=None*, **alternative_carrier*)
Initialize the record with a *version* number, an *error* information tuple, and zero or more *alternative_carrier*. The version number can be either an 8-bit integer (4-bit major, 4-bit minor), or a `'{major}.{minor}'` version string. An alternative carrier is given by a tuple with *carrier power state*, *carrier data reference* and

zero or more *auxiliary data references*. The *error* argument is a tuple with error reason and error data. Error information, if not None, is encoded as the local Error Record after all given alternative carriers.

**Parameters**

- **version** (*int or str*) – handover version number
- **error** (*tuple*) – error reason and data
- **alternative_carrier** (*tuple*) – alternative carrier entry

**type**
The Handover Select Record type is `urn:nfc:wkt:Hs`.

**name**
Value of the NDEF Record ID field, an empty `str` if not set.

**data**
A `bytes` object containing the NDEF Record PAYLOAD encoded from the current attributes.

**hexversion**
The version as an 8-bit integer with 4-bit major and minor part. This is a read-only attribute.

**version_info**
The version as a named tuple with major and minor version number attributes. This is a read-only attribute.

**version_string**
The version as the '{major}.{minor}' formatted string. This is a read-only attribute.

**error**
Either error information or None. Error details can be accessed with `error.error_reason` and `error.error_data`. Formatted error information is provided with `error.error_reason_string`.

**set_error(error_reason, error_data):**
Set error information. The *error_reason* argument is an 8-bit integer value but only values 1, 2 and 3 are defined in the specification. For defined error reasons the *error_data* argument is the associated value (which is a number in all cases). For undefined error reason values the *error_data* argument is `bytes`. Error reason value 0 is strictly reserved and never encoded or decoded.

**alternative_carriers**
A `list` of alternative carriers with attributes carrier_power_state, carrier_data_reference, and auxiliary_data_reference list.

**add_alternative_carrier(cps, cdr, *adr):**
Add a reference to a carrier data record within the handover select message. The carrier data reference *cdr* is the name (NDEF Record ID) of the carrier data record. The carrier power state *cps* is either 'inactive', 'active', 'activating', or 'unknown'. Any number of auxiliary data references *adr* may be added to link with other records in the message that carry information related to the carrier.

### 2.5.3 Handover Mediation Record

The Handover Mediation Record is the first record of a connection handover mediation message. Information enclosed within the payload of a handover mediation record includes the version number and zero or more references to alternative carrier information records subsequently encoded in the same message.

```
>>> import ndef
>>> carrier = ndef.Record('mimetype/subtype', 'ref', b'1234')
>>> message = [ndef.HandoverMediationRecord('1.3'), carrier]
>>> message[0].add_alternative_carrier('active', carrier.name)
```

**class HandoverMediationRecord**(*version='1.3'*, **alternative_carrier*)

Initialize the record with *version* number and zero or more *alternative_carrier*. The version number can be either an 8-bit integer (4-bit major, 4-bit minor), or a `'{major}.{minor}'` version string. An alternative carrier is given by a tuple with *carrier power state*, *carrier data reference* and zero or more *auxiliary data references*.

**Parameters**

- **version** (*int or str*) – handover version number
- **alternative_carrier** (*tuple*) – alternative carrier entry

**type**

The Handover Select Record type is `urn:nfc:wkt:Hm`.

**name**

Value of the NDEF Record ID field, an empty `str` if not set.

**data**

A `bytes` object containing the NDEF Record PAYLOAD encoded from the current attributes.

**hexversion**

The version as an 8-bit integer with 4-bit major and minor part. This is a read-only attribute.

**version_info**

The version as a named tuple with major and minor version number attributes. This is a read-only attribute.

**version_string**

The version as the '{major}.{minor}' formatted string. This is a read-only attribute.

**alternative_carriers**

A `list` of alternative carriers with attributes carrier_power_state, carrier_data_reference, and auxiliary_data_reference list.

**add_alternative_carrier(cps, cdr, *adr):**

Add a reference to a carrier data record within the handover mediation message. The carrier data reference *cdr* is the name (NDEF Record ID) of the carrier data record. The carrier power state *cps* is either 'inactive', 'active', 'activating', or 'unknown'. Any number of auxiliary data references *adr* may be added to link with other records in the message that carry information related to the carrier.

### 2.5.4 Handover Initiate Record

The Handover Initiate Record is the first record of a connection handover initiate message. Information enclosed within the payload of a handover initiate record includes the version number and zero or more references to alternative carrier information records subsequently encoded in the same message.

```
>>> import ndef
>>> carrier = ndef.Record('mimetype/subtype', 'ref', b'1234')
>>> message = [ndef.HandoverInitiateRecord('1.3'), carrier]
>>> message[0].add_alternative_carrier('active', carrier.name)
```

**class HandoverInitiateRecord**(*version='1.3'*, **alternative_carrier*)

Initialize the record with *version* number and zero or more *alternative_carrier*. The version number can be either an 8-bit integer (4-bit major, 4-bit minor), or a `'{major}.{minor}'` version string. An alternative carrier is given by a tuple with *carrier power state*, *carrier data reference* and zero or more *auxiliary data references*.

**Parameters**

- **version** (*int or str*) – handover version number

- **alternative_carrier** (*tuple*) – alternative carrier entry

**type**
The Handover Select Record type is `urn:nfc:wkt:Hi`.

**name**
Value of the NDEF Record ID field, an empty `str` if not set.

**data**
A `bytes` object containing the NDEF Record PAYLOAD encoded from the current attributes.

**hexversion**
The version as an 8-bit integer with 4-bit major and minor part. This is a read-only attribute.

**version_info**
The version as a named tuple with major and minor version number attributes. This is a read-only attribute.

**version_string**
The version as the '{major}.{minor}' formatted string. This is a read-only attribute.

**alternative_carriers**
A `list` of alternative carriers with attributes carrier_power_state, carrier_data_reference, and auxiliary_data_reference list.

**add_alternative_carrier(cps, cdr, *adr):**
Add a reference to a carrier data record within the handover initiate message. The carrier data reference *cdr* is the name (NDEF Record ID) of the carrier data record. The carrier power state *cps* is either 'inactive', 'active', 'activating', or 'unknown'. Any number of auxiliary data references *adr* may be added to link with other records in the message that carry information related to the carrier.

### 2.5.5 Handover Carrier Record

The Handover Carrier Record allows a unique identification of an alternative carrier technology in a handover request message when no carrier configuration data is to be provided. If the handover selector device has the same carrier technology available, it would respond with a carrier configuration record with payload type equal to the carrier type (that is, the triples (TNF, TYPE_LENGTH, TYPE) and (CTF, CARRIER_TYPE_LENGTH, CARRIER_TYPE) match exactly).

```
>>> import ndef
>>> record = ndef.HandoverCarrierRecord('application/vnd.wfa.wsc')
>>> record.name = 'wlan'
>>> print(record)
NDEF Handover Carrier Record ID 'wlan' CARRIER 'application/vnd.wfa.wsc' DATA 0 byte
```

**class HandoverCarrierRecord**(*carrier_type*, *carrier_data=None*, *reference=None*)
Initialize the HandoverCarrierRecord with *carrier_type*, *carrier_data*, and a *reference* that sets the *Record.name* attribute. The carrier type has the same format as a record type name, i.e. the combination of NDEF Record TNF and TYPE that is used by the *Record.type* attribute. The carrier_data argument must be a valid `bytearray` initializer, or None.

**Parameters**

- **carrier_type** (*str*) – initial value of the *carrier_type* attribute
- **carrier_data** (*sequence*) – initial value of the *carrier_data* attribute
- **reference** (*str*) – initial value of the the *name* attribute

**type**
The Handover Select Record type is `urn:nfc:wkt:Hc`.

**name**
Value of the NDEF Record ID field, an empty `str` if not set. The *reference* init argument can also be used to set this value.

**data**
A `bytes` object containing the NDEF Record PAYLOAD encoded from the current attributes.

**carrier_type**
Get or set the carrier type as a *`Record.type`* formatted representation of the Handover Carrier Record CTF and CARRIER_TYPE fields.

**carrier_data**
Contents of the Handover Carrier Record CARRIER_DATA field as a `bytearray`. The attribute itself is read-only but the content may be modified or expanded.

## 2.6 Bluetooth Secure Simple Pairing

**Note:** This is "work in progress" towards version 0.3

New in version 0.3.

### 2.6.1 Introduction

Bluetooth Secure Simple Pairing (SSP) has been introduced in Bluetooth Core Specification Version 2.1 + EDR as a method by which two Bluetooth devices can establish secure communication. With Bluetooth Core Specification Version 4.0 this was extended to cover Bluetooth Low Energy devices.

Bluetooth Secure Simple Pairing defines four different association models, one of them is using an out-of-band channel such as NFC. Secure Simple Pairing introduced in Bluetooth Version 2.1 + EDR uses Elliptic Curve Diffie-Hellman with curve P-192. Bluetooth Version 4.1 added the Secure Connections feature, which upgraded Secure Simple Pairing to utilize the P-256 elliptic curve. In either case the out-of-band communication transfers a public key commitment through a 128-bit hash and randomizer prior to in-band public key exchange. Bluetooth BR/EDR key generation is performed in the Controller. Bluetooth Low Energy, introduced with Core Specification Version 4.0, uses a Security Manager component on the device host to generate keys. Three pairing methods - Just Works, Passkey Entry, and Out of Band - were initially defined with protection levels depending on the secrecy of temporary keys exchanged while pairing. Bluetooth Version 4.2 then added LE Secure Connections with the same pairing methods and P-256 based Elliptic Curve Diffie-Hellman as for BR/EDR Secure Connections.

Bluetooth pairing is the process of connecting with a Bluetooth devices that has been found by device discovery. The discovery process provides the identity of the other device. The pairing process then yields a shared secret that is used to derive encryption keys. There are four pairing methods: Numeric Comparision, Just Works, Passkey Entry, and Out of Band. Numeric Comparision protects against man-in-the-middle by having the user confirm equality of a six digit number displayed on both devices. Just Works is basically the same but the number is not shown for confirmation. Passkey Entry requires one device to have a keypad and the other to have a display. A number entered into the keypad is shown on the other device for confirmation. Out of Band uses some external communication means to ensure that key material exchanged in-band belongs to the adressed communication partner.

NFC is a perfect fit for an out-of-band communication channel for Bluetooth device pairing. NFC communication only starts when two devices are in very close proximity, literally touched to each other, but works without any discovery, device selection or confirmation steps. NFC is comparatively slow and it is not always convinient to keep proximity for a longer period of time. So Bluetooth is also a perfect fit for NFC when larger or longer data transfers are requested. From an NFC point of view this is *Connection Handover* with Bluetooth out-of-band data transmitted as an alternative carrier.

Connection Handover may be performed between two NFC Devices (negotiated handover) or one NFC Device and another device that has an NFC Tag attached (static handover). In negotiated handover, the NFC Device that wants to establish an alternative connection sends a Connection Handover Request and waits for a Connection Handover Select message. In static handover, the NFC Device reads a Connection Handover Select message from the NFC Tag.

#### Bluetooth BR/EDR Out-of-Band Data

Table 1: Bluetooth BR/EDR Secure Simple Pairing OOB Data

| Element | | Size | Description |
|---|---|---|---|
| OOB Data Length | | 2 | Total length of OOB data including the Length field |
| Bluetooth Address | | 6 | The 48-bit Bluetooth Device Address (MAC Address) |
| OOB Optional Data | | N | Additional OOB data as Extended Inquiry Response[1] |
| | EIR 0x02 or 0x03 — Incomplete or Complete List of 16-bit Service Class UUIDs | | |
| | EIR 0x04 or 0x05 — Incomplete or Complete List of 32-bit Service Class UUIDs | | |
| | EIR 0x06 or 0x07 — Incomplete or Complete List of 128-bit Service Class UUIDs | | |
| | EIR 0x08 or 0x09 — Shortened or Complete Bluetooth Local Name | | |
| | EIR 0x0D — Class of Device | | |
| | EIR 0x0E — Simple Pairing Hash C-192 | | |
| | EIR 0x0F — Simple Pairing Randomizer R-192 | | |
| | EIR 0x1D — Simple Pairing Hash C-256 | | |
| | EIR 0x1E — Simple Pairing Randomizer R-256 | | |
| | << Other EIR data types >> | | |

#### Bluetooth LE Out-of-Band Data

Table 2: Bluetooth AD Types for OOB Pairing over NFC

| AD Type | Significance | Description |
|---|---|---|
| 0x1B | Mandatory | LE Bluetooth Device Address |
| 0x1C | Mandatory | LE Role |
| 0x10 | Optional | Security Manager TK Value (LE legacy pairing) |
| 0x19 | Optional | Appearance |
| 0x01 | Optional | Flags |
| 0x08 or 0x09 | Optional | Shortened or Complete Bluetooth Local Name |
| 0x22 | Optional | LE Secure Connections Confirmation Value |
| 0x23 | Optional | LE Secure Connections Random Value |
| << Other AD types >> | | |

### 2.6.2 NDEF Records

**class** ndef.bluetooth.**BluetoothRecord**

A base class implementing dictionary-like EIR/AD data type access for the *BluetoothEasyPairingRecord* and the *BluetoothLowEnergyRecord*. It should not be used directly as an NDEF record type.

[1] Data elements within an Extended Inquiry Response are in no specific order. The order shown is only for illustration.

Dictionary-like access works with either numeric or text keys. Numeric keys are defined in Bluetooth Assigned Numbers under Generic Access Profile. Recognized text keys are the data type names that are given by *attribute_names*.

```
>>> import ndef
>>> dict_like = ndef.bluetooth.BluetoothRecord()
>>> dict_like[0x09] = b'Device Name'
>>> dict_like.get('Complete Local Name')
b'Device Name'
>>> dict_like.get('Shortened Local Name', b'default name')
b'default name'
>>> [dict_like.get(name) for name in dict_like.attribute_names if name in dict_
↪like]
[b'Device Name']
```

**attribute_names**

Returns all Bluetooth EIR/AD data type names that may be used as text keys. Note that 'Simple Pairing Hash C' and 'Simple Pairing Hash C-192' as well as 'Simple Pairing Randomizer R' and 'Simple Pairing Randomizer R-192' resolve to the same numeric key, respectively.

```
>>> import ndef
>>> print('\n'.join(sorted(ndef.bluetooth.BluetoothRecord().attribute_names)))
Appearance
Class of Device
Complete List of 128-bit Service Class UUIDs
Complete List of 16-bit Service Class UUIDs
Complete List of 32-bit Service Class UUIDs
Complete Local Name
Flags
Incomplete List of 128-bit Service Class UUIDs
Incomplete List of 16-bit Service Class UUIDs
Incomplete List of 32-bit Service Class UUIDs
LE Bluetooth Device Address
LE Role
LE Secure Connections Confirmation Value
LE Secure Connections Random Value
Manufacturer Specific Data
Security Manager Out of Band Flags
Security Manager TK Value
Shortened Local Name
Simple Pairing Hash C
Simple Pairing Hash C-192
Simple Pairing Hash C-256
Simple Pairing Randomizer R
Simple Pairing Randomizer R-192
Simple Pairing Randomizer R-256
```

### Easy Pairing Record

**class BluetoothEasyPairingRecord**(*device_address*, **eir*)

This class decodes and encodes Bluetooth BR/EDR Secure Simple Pairing Out-of-Band data and provides access to the embedded information.

A *BluetoothEasyPairingRecord* must be initialized with at least the Bluetooth Device Address as the first argument. Any following arguments are expected to be key-value tuples where the key may be an EIR data type number or a recognized data type name and the value must be a `bytes` object with the corresponding data type octets (in little endian order for multi-byte values)..

```
>>> import ndef
>>> eir_list = [(0x0D, b'\x04\x01\x12'), ('Shortened Local Name', b'My Blue')]
>>> record = ndef.BluetoothEasyPairingRecord('01:02:03:04:05:06', *eir_list)
>>> record['Incomplete List of 16-bit Service Class UUIDs'] = b'\x0A\x11'
>>> print(record)
NDEF Bluetooth Easy Pairing Record ID '' Attributes 0x02 0x08 0x0D
>>> octets = b''.join(ndef.message_encoder([record]))
>>> print(list(ndef.message_decoder(octets))[0])
NDEF Bluetooth Easy Pairing Record ID '' Attributes 0x02 0x08 0x0D
```

**type**
The read-only Bluetooth Easy Pairing Record type.

```
>>> record.type
'application/vnd.bluetooth.ep.oob'
```

**name**
Value of the NDEF Record ID field, an empty `str` if not set.

```
>>> record.name = 'Easy Pairing Record'
>>> record.name
'Easy Pairing Record'
```

**device_address**
The *DeviceAddress* decoded from or to be encoded into the out-of-band BD_ADDR field.

```
>>> record.device_address = '01:02:03:04:05:06'
>>> record.device_address
ndef.bluetooth.DeviceAddress('01:02:03:04:05:06', 'public')
```

**device_name**
Get or set the Bluetooth Local Name.

The Local Name, if configured on the Bluetooth device, is the name that may be displayed to the device user as part of the UI involving operations with Bluetooth devices. It may be encoded as either 'Complete Local name' or 'Shortened Local Name' EIR data type.

This attribute provides the Local Name as a text string. The value returned is the 'Complete Local Name' or 'Shortened Local Name' evaluated in that order. None is returned if neither EIR data type exists.

A device name assigned to this attribute is always stored as the 'Complete Local Name' and removes a 'Shortened Local Name' EIR data type if formerly present.

```
>>> record['Shortened Local Name'] = b'shortened name'
>>> record.device_name
'shortened name'
>>> record.device_name = "My \u2039BR/EDR\u203a Device"
>>> record.device_name
'My ‹BR/EDR› Device'
>>> assert record.get('Shortened Local Name') is None
>>> record['Complete Local Name']
b'My \xe2\x80\xb9BR/EDR\xe2\x80\xba Device'
```

**device_class**
Get or set the Bluetooth Class of Device information. Reading returns a *DeviceClass* object. The attribute may be set to either a *DeviceClass* object or the 24-bit Class of Device integer value. If the Bluetooth Class of Device EIR data type is not present when reading, the attribute is `ndef.bluetooth.DeviceClass(0x000000)`.

```
>>> record.device_class
ndef.bluetooth.DeviceClass(0x120104)
>>> ndef.bluetooth.DeviceClass.decode(record.get('Class of Device'))
ndef.bluetooth.DeviceClass(0x120104)
>>> record.device_class = 0x120104
```

**service_class_list**
A read-only list of *ServiceClass* instances build from all available Bluetooth Service Class UUID attributes (complete/incomplete and 16/32/128 bit EIR/AD types).

```
>>> record.service_class_list
[ndef.bluetooth.ServiceClass('0000110a-0000-1000-8000-00805f9b34fb')]
```

**add_service_class**(*service_class*, *complete=False*)
Add a *service_class* identifier and set the resulting list of 16, 32 or 128 bit Service Class UUIDs to either *complete* or incomplete. The *service_class* argument must be a *ServiceClass* or an initializer thereof.

```
>>> assert 'Incomplete List of 16-bit Service Class UUIDs' in record
>>> assert 'Complete List of 16-bit Service Class UUIDs' not in record
>>> record.add_service_class(0x110B, complete=True)
>>> assert 'Incomplete List of 16-bit Service Class UUIDs' not in record
>>> assert 'Complete List of 16-bit Service Class UUIDs' in record
>>> [sc.name for sc in record.service_class_list]
['Audio Source', 'Audio Sink']
```

**simple_pairing_hash_192**
Get or set the Simple Pairing Hash C-192.

The Simple Pairing Hash C-192 is a commitment of the device's public key computed as HMAC-SHA-256 for the Curve-192 ECPK and Randomizer R-192. The Hash C should be generated anew for each pairing.

This attribute returns either the 128-bit integer converted from the 16-octet 'Simple Pairing Hash C-192' EIR value or None if the EIR data type is not present. When set, it stores a 128-bit integer as the 16-octet value of the 'Simple Pairing Hash C-192' EIR data type.

```
>>> record.simple_pairing_hash_192 = 0x1234567890ABCDEF1234567890ABCDEF
>>> record.get('Simple Pairing Hash C-192').hex()
'efcdab9078563412efcdab9078563412'
```

**simple_pairing_randomizer_192**
Get or set the Simple Pairing Randomizer R-192.

If both devices transmit and receive data over NFC, then mutual authentication is based on the commitments of the public keys by Hash C exchanged out-of-band. If one device can only send information (typically an NFC Tag that is read by the other device), then authentication of the reading device will be based on that device knowing a random number R read from the NFC Tag. In this case, R must be secret: it can be created afresh every time (if the NFC Tag content can be modified by the host), or access to the device sending R must be restricted. Generally, if R is not sent by a device it is assumed to be 0 by the device receiving the out-of-band information.

The Simple Pairing Randomizer R-192 is used with P192 Elliptic Curve Diffie Hellmann.

This attribute returns either the 128-bit integer converted from the 16-octet 'Simple Pairing Randomizer R-192' EIR value or None if the EIR data type is not present. When set, it stores a 128-bit integer as the 16-octet value of the 'Simple Pairing Randomizer R-192' EIR data type.

```
>>> record.simple_pairing_randomizer_192 = 0x0102030405060708090A0B0C0D0E0F
>>> record.get('Simple Pairing Randomizer R-192').hex()
'0f0e0d0c0b0a09080706050403020100'
```

**simple_pairing_hash_256**
Get or set the Simple Pairing Hash C-256.

The Simple Pairing Hash C-256 is a commitment of the device's public key computed as HMAC-SHA-256 for the Curve-256 ECPK and Randomizer R-256. The Hash C should be generated anew for each pairing.

This attribute returns either the 128-bit integer converted from the 16-octet 'Simple Pairing Hash C-256' EIR value or None if the EIR data type is not present. When set, it stores a 128-bit integer as the 16-octet value of the 'Simple Pairing Hash C-256' EIR data type.

```
>>> record.simple_pairing_hash_256 = 0x1234567890ABCDEF1234567890ABCDEF
>>> record.get('Simple Pairing Hash C-256').hex()
'efcdab9078563412efcdab9078563412'
```

**simple_pairing_randomizer_256**
Get or set the Simple Pairing Randomizer R-256.

If both devices transmit and receive data over NFC, then mutual authentication is based on the commitments of the public keys by Hash C exchanged out-of-band. If one device can only send information (typically an NFC Tag that is read by the other device), then authentication of the reading device will be based on that device knowing a random number R read from the NFC Tag. In this case, R must be secret: it can be created afresh every time (if the NFC Tag content can be modified by the host), or access to the device sending R must be restricted. Generally, if R is not sent by a device it is assumed to be 0 by the device receiving the out-of-band information.

The Simple Pairing Randomizer R-256 is used with P256 Elliptic Curve Diffie Hellmann.

This attribute returns either the 128-bit integer converted from the 16-octet 'Simple Pairing Randomizer R-256' EIR value or None if the EIR data type is not present. When set, it stores a 128-bit integer as the 16-octet value of the 'Simple Pairing Randomizer R-256' EIR data type.

```
>>> record.simple_pairing_randomizer_256 = 0x0102030405060708090A0B0C0D0E0F
>>> record.get('Simple Pairing Randomizer R-256').hex()
'0f0e0d0c0b0a09080706050403020100'
```

### Low Energy Record

**class BluetoothLowEnergyRecord**(*device_address*, **advertising_data*)

```
>>> import ndef
>>> record = ndef.BluetoothLowEnergyRecord((0x08, b'My Blue'), (0x0D, b'100420'))
>>> print(record)
NDEF Bluetooth Low Energy Record ID '' Attributes 0x08 0x0D
```

**type**
The read-only Bluetooth Low Energy Record type.

```
>>> record.type
'application/vnd.bluetooth.le.oob'
```

**name**
Value of the NDEF Record ID field, an empty `str` if not set.

```
>>> record.name = 'BLE Record'
>>> record.name
'BLE Record'
```

**device_address**

Get or set the LE Bluetooth Device Address.

The LE Bluetooth Device Address data value consists of 7 octets made up from the 48 bit address that is used for Bluetooth pairing over the LE transport and a flags octet that defines the address type. The address type distinguishes a Public Device Address versus a Random Device Address. A Random Device Address sent with BLE out-of-band data should be used on the LE transport for at least ten minutes after the NFC data exchange.

This attribute returns a *DeviceAddress* or None, depending on whether the 'LE Bluetooth Device Address' AD type is present or not (under rare circumstances or just by failure it may not be). The *device_address* attribute may be set by assigning it another *DeviceAddress*, a tuple of address and address type strings, or a sole address string which implies a public address type.

```
>>> record.device_address = '01:02:03:04:05:06'
>>> record.device_address
ndef.bluetooth.DeviceAddress('01:02:03:04:05:06', 'public')
>>> record.device_address = ('01:02:03:04:05:06', 'random')
>>> record.device_address
ndef.bluetooth.DeviceAddress('01:02:03:04:05:06', 'random')
```

**device_name**

Get or set the Bluetooth Local Device Name.

The Local Name, if configured on the Bluetooth device, is the name that may be displayed to the device user as part of the UI involving operations with Bluetooth devices. It may be encoded as either 'Complete Local name' or 'Shortened Local Name' AD type.

This attribute provides the Local Name as a text string. The value returned is the 'Complete Local Name' or 'Shortened Local Name' evaluated in that order. None is returned if neither AD type exists.

A device name assigned to this attribute is always stored as the 'Complete Local Name' and removes a 'Shortened Local Name' AD type if formerly present.

```
>>> record['Shortened Local Name'] = b'shortened name'
>>> record.device_name
'shortened name'
>>> record.device_name = "My \u2039BLE\u203a Device"
>>> record.device_name
'My ‹BLE› Device'
>>> assert record.get('Shortened Local Name') is None
>>> record.get('Complete Local Name')
b'My \xe2\x80\xb9BLE\xe2\x80\xba Device'
```

**appearance**

Get or set the representation of the external appearance of the device, used by the discovering device to represent an icon, string, or similar to the user. The returned value is a tuple with the numeric value and a textual description, or None if the 'Appearance' AD type is not found. The appearance attribute accepts either a numeric value or a description string.

Appearance strings consist of a generic category and an optional subtype. If a subtype is present it follows the generic category text after a colon.

```
>>> record['Appearance'] = b'\x81\x03'
>>> print(record.appearance)
(897, 'Blood Pressure: Arm')
>>> print("category '{0[0]}' subtype '{0[1]}'".format(record.appearance[1].
↪split(': ')))
category 'Blood Pressure' subtype 'Arm'
>>> record.appearance = "Thermometer"
>>> record['Appearance']
b'\x00\x03'
>>> record.appearance = 0x0280
>>> print(record.appearance)
(640, 'Media Player')
```

**appearance_strings**

A list of all known appearance strings that may be assigned to *appearance*.

```
>>> print('\n'.join(record.appearance_strings))
Unknown
Phone
Computer
Watch
Watch: Sports Watch
Clock
Display
Remote Control
Eye-glasses
Tag
Keyring
Media Player
Barcode Scanner
Thermometer
Thermometer: Ear
Heart Rate Sensor
Heart Rate Sensor: Belt
Blood Pressure
Blood Pressure: Arm
Blood Pressure: Wrist
Human Interface Device
Human Interface Device: Keyboard
Human Interface Device: Mouse
Human Interface Device: Joystick
Human Interface Device: Gamepad
Human Interface Device: Digitizer Tablet
Human Interface Device: Card Reader
Human Interface Device: Digital Pen
Human Interface Device: Barcode Scanner
Glucose Meter
Running Walking Sensor
Running Walking Sensor: In-Shoe
Running Walking Sensor: On-Shoe
Running Walking Sensor: On-Hip
Cycling
Cycling: Cycling Computer
Cycling: Speed Sensor
Cycling: Cadence Sensor
Cycling: Power Sensor
Cycling: Speed and Cadence Sensor
```

(continues on next page)

(continued from previous page)

```
Pulse Oximeter
Pulse Oximeter: Fingertip
Pulse Oximeter: Wrist Worn
Weight Scale
Outdoor Sports
Outdoor Sports: Location Display Device
Outdoor Sports: Location and Navigation Display Device
Outdoor Sports: Location Pod
Outdoor Sports: Location and Navigation Pod
```

**role_capabilities**

Get or set the LE role capabilities of the device. The value is a string describing one of the four defined roles `Peripheral`, `Central`, `Peripheral/Central` (Peripheral Role preferred for connection establishment), or `Central/Peripheral` (Central is preferred for connection establishment).

```
>>> record['LE Role'] = b'\x02'
>>> print(record.role_capabilities)
Peripheral/Central
>>> record.role_capabilities = "Central"
>>> assert record['LE Role'] == b'\x01'
```

**flags**

Get or set the Flags bitmap.

The 'Flags' AD type contains information on which discoverable mode to use and BR/EDR support and capability. The attribute returns the numerical flags value and descriptions for raised bits as an N-tuple. The attribute accepts either a numerical flags value or a tuple of description strings.

```
>>> record['Flags'] = b'\x05'
>>> print(record.flags)
(5, 'LE Limited Discoverable Mode', 'BR/EDR Not Supported')
>>> record.flags = ("LE General Discoverable Mode",)
>>> record['Flags']
b'\x02'
>>> record.flags = 8
>>> print(record.flags)
(8, 'Simultaneous LE and BR/EDR to Same Device Capable (Controller)')
```

**security_manager_tk_value**

Get or set the Security Manager TK Value.

The Security Manager TK Value is used by the LE Security Manager in the OOB association model with LE Legacy pairing. Reading this attribute returns an unsigned integer converted from the 16 byte 'Security Manager TK Value' AD type octets, or None if the AD type is not found. An unsigned integer assigned to this attribute is written as the 16 byte 'Security Manager TK Value' AD type after conversion.

```
>>> record.security_manager_tk_value = 0x1234567890ABCDEF1234567890ABCDEF
>>> record.get('Security Manager TK Value').hex()
'efcdab9078563412efcdab9078563412'
>>> record.security_manager_tk_value
24197857200151252728969465429440056815
```

**secure_connections_confirmation_value**

Get or set the LE Secure Connections Confirmation Value.

The LE Secure Connections Confirmation Value is used by the LE Security Manager if the OOB association model with LE Secure Connections pairing is used. Reading this attribute returns an unsigned integer

converted from the 16 byte 'LE Secure Connections Confirmation Value' AD type octets, or None if the AD type is not found. An unsigned integer assigned to this attribute is written as the 16 byte 'LE Secure Connections Confirmation Value' AD type after conversion.

```
>>> record.secure_connections_confirmation_value =␣
↪0x1234567890ABCDEF1234567890ABCDEF
>>> record.get('LE Secure Connections Confirmation Value').hex()
'efcdab9078563412efcdab9078563412'
>>> record.secure_connections_confirmation_value
24197857200151252728969465429440056815
```

**secure_connections_random_value**
Get the LE Secure Connections Random Value.

The LE Secure Connections Random Value is used by the LE Security Manager if the OOB association model with LE Secure Connections pairing is used. Reading this attribute returns an unsigned integer converted from the 16 byte 'LE Secure Connections Random Value' AD type octets, or None if the AD type is not found. An unsigned integer assigned to this attribute is written as the 16 byte 'LE Secure Connections Random Value' AD type after conversion.

```
>>> record.secure_connections_random_value =␣
↪0x1234567890ABCDEF1234567890ABCDEF
>>> record.get('LE Secure Connections Random Value').hex()
'efcdab9078563412efcdab9078563412'
>>> record.secure_connections_random_value
24197857200151252728969465429440056815
```

### 2.6.3 Data Types

#### Device Address

**class** ndef.bluetooth.**DeviceAddress**(*address*, *address_type='public'*)
Representation of a Bluetooth device address, either initialized with *address* and *address_type* or decoded from octets. The *address* argument for initialization is a MAC address string with colons or dashes as separators. The default *address_type* is 'public', for a Bluetooth LE address it may be set to 'random'. Note that this only makes a difference when encoding.

```
>>> import ndef
>>> print(ndef.bluetooth.DeviceAddress('01:02:03:04:05:06'))
Device Address 01:02:03:04:05:06 (public)
```

**static decode**(*octets*)
Returns a *DeviceAddress* instance constructed from either a BD_ADDR (6 octets) or 'LE Bluetooth Device Address' (7 octets).

```
>>> ndef.bluetooth.DeviceAddress.decode(b'\x06\x05\x04\x03\x02\x01')
ndef.bluetooth.DeviceAddress('01:02:03:04:05:06', 'public')
>>> ndef.bluetooth.DeviceAddress.decode(b'\x06\x05\x04\x03\x02\x01\x01')
ndef.bluetooth.DeviceAddress('01:02:03:04:05:06', 'random')
```

**encode**(*context='LE'*)
Returns the Bluetooth address as bytes in little endian order. The *context* argument determines the encoding format. For a Bluetooth LE address seven bytes are returned and the last byte discriminates between a public or random address. For BD_ADDR encoding the *context* must be 'EP' (for Easy Pairing).

```
>>> ndef.bluetooth.DeviceAddress('01:02:03:04:05:06').encode('EP')
b'\x06\x05\x04\x03\x02\x01'
>>> ndef.bluetooth.DeviceAddress('01:02:03:04:05:06').encode('LE')
b'\x06\x05\x04\x03\x02\x01\x00'
```

**addr**
Get or set the Bluetooth Device Address. The address is a string in typical MAC address notation, both `:` and `-` are acceptable delimiters.

```
>>> bdaddr = ndef.bluetooth.DeviceAddress('01:02:03:04:05:06')
>>> bdaddr.addr
'01:02:03:04:05:06'
>>> bdaddr.addr = '06-05-04-03-02-01'
>>> bdaddr.addr
'06:05:04:03:02:01'
```

**type**
Get or set the Bluetooth LE address type which may be either 'public' or 'random'.

```
>>> bdaddr = ndef.bluetooth.DeviceAddress('01:02:03:04:05:06', 'public')
>>> bdaddr.type = 'random'
>>> bdaddr
ndef.bluetooth.DeviceAddress('01:02:03:04:05:06', 'random')
```

### Device Class

**class** ndef.bluetooth.**DeviceClass**(*cod*)
Mapping of the Bluetooth 'Class of Device' information. An instance can be created with an integer argument that represents the 24 bits of the Class of Device structure, or by decoding a 3-byte sequence with the 24 bits in transmission order (little endian).

```
>>> import ndef
>>> print(ndef.bluetooth.DeviceClass(0x120104))
Device Class Computer - Desktop workstation - Networking and Object Transfer
```

**static decode**(*octets*)
Returns a *DeviceClass* instance with the 24 bits 'Class of Device' information decoded from *octets*. The *octets* argument must be a `bytes` or `bytearray` object of length 3 and in little endian order.

```
>>> ndef.bluetooth.DeviceClass.decode(b'\x04\x01\x12')
ndef.bluetooth.DeviceClass(0x120104)
```

**encode**()
Returns 3 `bytes` with the 'Class of Device' integer in little endian order.

```
>>> ndef.bluetooth.DeviceClass(0x120104).encode()
b'\x04\x01\x12'
```

**major_device_class**
The major device class string (read-only).

```
>>> ndef.bluetooth.DeviceClass(0x120104).major_device_class
'Computer'
```

**minor_device_class**
The minor device class string (read-only).

```
>>> ndef.bluetooth.DeviceClass(0x120104).minor_device_class
'Desktop workstation'
```

**major_service_class**
A tuple of major service class strings (read-only).

```
>>> ndef.bluetooth.DeviceClass(0x120104).major_service_class
('Networking', 'Object Transfer')
```

### Service Class

**class** ndef.bluetooth.**ServiceClass**(**args*, ***kwargs*)
The ServiceClass represents a single Bluetooth Service Class UUID. The first positional argument may be a Bluetooth 'uuid16' or 'uuid32' integer, a Bluetooth service class name, or any of the UUID string formats accepted by `uuid.UUID`. Alternatively, the same keyword arguments supported by `uuid.UUID` may be used.

```
>>> import ndef
>>> ndef.bluetooth.ServiceClass(0x110A)
ndef.bluetooth.ServiceClass('0000110a-0000-1000-8000-00805f9b34fb')
>>> ndef.bluetooth.ServiceClass("Audio Source")
ndef.bluetooth.ServiceClass('0000110a-0000-1000-8000-00805f9b34fb')
```

**static decode**(*octets*)
Returns a *ServiceClass* instance decoded from *octets*. The *octets* argument must be a `bytes` or `bytearray` object of either length 2, 4, or 16 in little endian order.

```
>>> ndef.bluetooth.ServiceClass.decode(b'\x0A\x11')
ndef.bluetooth.ServiceClass('0000110a-0000-1000-8000-00805f9b34fb')
```

**encode**()
Return the `bytes` representation of the Service Class UUID in little endian order. The number of octets is 2 or 4 for a Bluetooth 'uuid16' or 'uuid32' and 16 for any other UUID value.

```
>>> ndef.bluetooth.ServiceClass(0x110A).encode()
b'\n\x11'
>>> ndef.bluetooth.ServiceClass(0x1000110A).encode()
b'\n\x11\x00\x10'
```

**uuid**
A `uuid.UUID` object that represents the Bluetooth Service Class UUID (read-only).

```
>>> ndef.bluetooth.ServiceClass(0x110A).uuid
UUID('0000110a-0000-1000-8000-00805f9b34fb')
```

**name**
The Bluetooth Service Class UUID name (read-only). Depending on the UUID value this is either one of `names` or the UUID string representation.

```
>>> ndef.bluetooth.ServiceClass(0x110A).name
'Audio Source'
>>> ndef.bluetooth.ServiceClass(0x1000110A).name
'1000110a-0000-1000-8000-00805f9b34fb'
```

**static get_uuid_names**()
Returns a tuple of all known Bluetooth Service Class UUID names.

```
>>> print('\n'.join(sorted(ndef.bluetooth.ServiceClass.get_uuid_names())))
A/V Remote Control
A/V Remote Control Controller
A/V Remote Control Target
Advanced Audio Distribution
Audio Sink
Audio Source
Basic Imaging Profile
Basic Printing
Browse Group Descriptor
Common ISDN Access
Cordless Telephony
Dialup Networking
Direct Printing
Direct Printing Reference
ESDP UPNP IP LAP
ESDP UPNP IP PAN
ESDP UPNP L2CAP
Fax
GN
GNSS
GNSS Server
Generic Audio
Generic File Transfer
Generic Networking
Generic Telephony
HCR Print
HCR Scan
HDP
HDP Sink
HDP Source
Handsfree
Handsfree Audio Gateway
Hardcopy Cable Replacement
Headset
Headset - Audio Gateway (AG)
Headset - HS
Human Interface Device
Imaging Automatic Archive
Imaging Referenced Objects
Imaging Responder
Intercom
IrMC Sync
IrMC Sync Command
LAN Access Using PPP
Message Access Profile
Message Access Server
Message Notification Server
NAP
OBEX File Transfer
OBEX Object Push
PANU
Phonebook Access
Phonebook Access - PCE
Phonebook Access - PSE
PnP Information
Printing Status
```

(continues on next page)

(continued from previous page)

```
Reference Printing
Reflected UI
SIM Access
Serial Port
Service Discovery Server
UPNP IP Service
UPNP Service
Video Distribution
Video Sink
Video Source
WAP
WAP Client
```

## 2.7 Wi-Fi Simple Configuration

New in version 0.2.

### 2.7.1 Overview

The Wi-Fi Alliance developed the Wi-Fi Simple Configuration specification to simplify the security setup and management of wireless networks. It is branded as Wi-Fi Protected Setup and can be used in traditional infrastructure networks as well as with Wi-Fi Direct. One of the three Wi-Fi Protected Setup methods uses NFC as an out-of-band channel to provision Wi-Fi devices with the network credentials (see also this short intro). All details can be learned from the Wi-Fi Alliance specifications.

The Wi-Fi Simple Configuration NFC out-of-band interface provides three usage models for provisioning an *Enrollee*, a device seeking to join a WLAN domain, with WLAN credentials. Devices with the authority to issue and revoke credentials are termed *Registrar*. A Registrar may be integrated into an Access Point.

*Password Token*

> A Password Token carries an Out-of-Band Device Password from an Enrollee to an NFC-enabled Registrar device. The device password is then used with the Wi-Fi in-band registration protocol to provision network credentials; an NFC Interface on the Enrollee is not required.

*Configuration Token*

> A Configuration Token carries unencrypted credential from an NFC-enabled Registrar to an NFC-enabled Enrollee device. A Configuration Token is created when the user touches the Registrar to retrieve the current network settings and allows subsequent configuration of one or more Enrollees.

*Connection Handover*

> Connection Handover is a protocol run between two NFC Peer Devices to establish an alternative carrier connection. The Connection Handover protocol is defined by the NFC Forum. Together with Wi-Fi Simple Configuration it helps connect to a Wi-Fi Infrastructure Access Point or a Wi-Fi Direct Group Owner.

#### Password Token

A Wi-Fi Password Token carries an NDEF Record with Payload Type “application/vnd.wfa.wsc” that contains an Out-Of-Band Device Password from the Enrollee. When presented to an NFC-enabled Registrar, typically an Access Point, the Wi-Fi in-band registration protocol uses the device password from the Password Token, instead of requiring

the user to manually input a password. Compared to manual input, a Password Token increases the effective security strength of the registration protocol by allowing for longer passwords and no need for key pad compatible characters.

The contents of a Wi-Fi Password Token are shown below. A parser must not rely on any specific order of the attributes, the order shown is only representational. Wi-Fi Attributes are encoded in the Wi-Fi Simple Configuration TLV Data Format (a Type-Length-Value format with 16-bit Type and 16-bit Length fields).

| Attribute | | Required/Conditional/Optional \| Description | |
|---|---|---|---|
| OOB Device Password | | R | A TLV with fixed data structure[1] |
| | Public Key Hash | R | The Enrollee's public key hash[2] |
| | Password ID | R | A 16 bit identifier for the device password[4] |
| | Device Password | R | The zero or 16–32 octet long device password[3] |
| WFA Vendor Extension | | C | Vendor Extension with Vendor ID 00:37:2A[5] |
| | Version2 | C | Wi-Fi Simple Configuration version[6] |
| | <other . . . > | O | Other WFA Vendor Extension subelements |
| <other . . . > | | O | Other Wi-Fi Simple Configuration TLVs |

**Example**

```
>>> import ndef
>>> import random
>>> import hashlib
>>> pkhash = hashlib.sha256(b'my public key goes here').digest()[0:20]
>>> pwd_id = random.randint(16, 65535)
>>> my_pwd = b"long password can't guess"
>>> oobpwd = ndef.wifi.OutOfBandPassword(pkhash, pwd_id, my_pwd)
>>> wfaext = ndef.wifi.WifiAllianceVendorExtension((0, b'\x20'))
>>> record = ndef.WifiSimpleConfigRecord()
>>> record.name = 'my password token'
>>> record['oob-password'] = [oobpwd.encode()]
>>> record['vendor-extension'] = [wfaext.encode()]
>>> print(record)
NDEF Wifi Simple Config Record ID 'my password token' Attributes 0x102C 0x1049
>>> octets = b''.join(ndef.message_encoder([record]))
>>> len(octets)
105
```

### Configuration Token

A Wi-Fi Configuration Token carries an NDEF Record with Payload Type "application/vnd.wfa.wsc" that contains unencrypted credential(s) issued by an NFC-enabled Registrar. An NFC-enabled Enrollee uses the credential(s) to directly connect to the Wi-Fi network without the need to run the Wi-Fi Simple Configuration registration protocol.

[1] The Out-Of-Band Device Password is a fixed data structure with three fields. The public key hash is in the first 20 octets. The password id uses the next 2 octets. The remaining TLV Length minus 22 octets contain the device password. The device passowrd must be at least 16 and at most 32 octets.

[2] The Public Key Hash field contains the first 160 bits of the SHA-256 hash of the Enrollee's public key that will be transmitted with message M1 of the registration protocol.

[4] The Password ID is an arbitrarily-selected number between 0x0010 and 0xFFFF. During the in-band registration protocol the Registrar sends the Password ID back to the Enrollee to identify the device password that is being used.

[3] The Device Password is zero length (absent) when used in negotiated connection handover between two Wi-Fi Peer To Peer devices, in which case the Password ID is equal to *NFC-Connection-Handover* (0x0007).

[5] The Wi-Fi Alliance Vendor Extension is a Vendor Extension Attribute with the first three octets (the Vendor ID) set to 00:37:2A (the Wi-Fi Alliance OUI). The remaining octets hold WFA Vendor Extension sub-elements in a Type-Length-Value format with 8-bit Type and 8-bit Length fields.

[6] The Version2 Attribute contains the Wi-Fi Simple Configuration version in a 1-octet field. The octet is split into the major version number in the most significant 4 bits and the minor version number in the least significant 4 bits. The Attribute is encoded as a WFA Vendor Extension sub-element with ID 0x00 and Length 0x01.

The contents of a Wi-Fi Configuration Token are shown below. A parser must not rely on any specific order of the attributes, the order shown is only representational. Wi-Fi Attributes are encoded in the Wi-Fi Simple Configuration TLV Data Format (a Type-Length-Value format with 16-bit Type and 16-bit Length fields).

| Attribute | | | Required/Conditional/Optional \| Description | |
|---|---|---|---|---|
| Credential | | | R | A single WLAN credential[7] |
| | Network Index | | R | Deprecated – always set to 1. |
| | SSID | | R | Network name (802.11 service set identifier). |
| | Authentication Type | | R | Network authentication type. |
| | Encryption Type | | R | Encryption capabilities. |
| | Network Key | | R | Encryption Key. |
| | MAC Address | | R | Enrollee's or broadcast MAC address[8] |
| | WFA Vendor Extension | | O | Vendor Extension with WFA Vendor ID 00:37:2A |
| | | Key Sharable | O | Whether the key may be shared with other devices |
| | | <other . . . > | O | Other WFA Vendor Extension subelements |
| | <other . . . > | | O | Other Wi-Fi Simple Configuration TLVs |
| RF Bands | | | O | Operating band of the AP or P2P group owner.[9] |
| RF Channel | | | O | Operating channel of AP or P2P group owner.[9] |
| MAC Address | | | O | The BSSID of the AP or Wi-Fi P2P group owner.[9] |
| WFA Vendor Extension | | | C | Vendor Extension with Vendor ID 00:37:2A[5] |
| | Version2 | | C | Wi-Fi Simple Configuration version[6] |
| | <other . . . > | | O | Other WFA Vendor Extension subelements |
| <other . . . > | | | O | Other Wi-Fi Simple Configuration TLVs |

**Example**

```
>>> import ndef
>>> credential = ndef.wifi.Credential()
>>> credential.set_attribute('network-index', 1)
>>> credential.set_attribute('ssid', b'my network name')
>>> credential.set_attribute('authentication-type', 'WPA2-Personal')
>>> credential.set_attribute('encryption-type', 'AES')
>>> credential.set_attribute('network-key', b'my secret password')
>>> credential.set_attribute('mac-address', b'\xFF\xFF\xFF\xFF\xFF\xFF')
>>> wfa_ext = ndef.wifi.WifiAllianceVendorExtension()
>>> wfa_ext.set_attribute('network-key-shareable', 1)
>>> credential['vendor-extension'] = [wfa_ext.encode()]
>>> print(credential)
Credential Attributes 0x1003 0x100F 0x1020 0x1026 0x1027 0x1045 0x1049
>>> record = ndef.wifi.WifiSimpleConfigRecord()
>>> record.name = 'my config token'
>>> record.set_attribute('credential', credential)
```

(continues on next page)

[7] The Credential is a compound attribute that contains other Wi-Fi Simple Configuration TLVs. A parser must not assume any specific order of the enclosed data elements.

[8] This should be the Enrollee's MAC address if the credential was specifically issued and will be valid only for the device with this MAC address. This can only be if the Registrar has prior knowledge of the Enrollee's MAC address and it's only effective if the AP is also able to restrict use of the credential to the provisioned device. In any other case the broadcast MAC address should be used.

[9] The optional RF Bands, AP Channel and MAC Address attributes may be included as hints to help the Station/Enrollee to find the AP without a full scan. It is recommended to include those attributes if known. If the RF Bands attribute and AP Channel attribute are both included then the RF Bands attribute indicates the band that the channel specified by the AP Channel attribute is in. If the RF Bands attribute is included without the AP Channel attribute then it indicates the RF Bands in which the AP is operating with the network name specified by the SSID attribute in the Credential.

(continued from previous page)

```
>>> record.set_attribute('rf-bands', ('2.4GHz', '5.0GHz'))
>>> wfa_ext = ndef.wifi.WifiAllianceVendorExtension()
>>> wfa_ext.set_attribute('version-2', 0x20)
>>> record['vendor-extension'] = [wfa_ext.encode()]
>>> print(record)
NDEF Wifi Simple Config Record ID 'my config token' Attributes 0x100E 0x103C 0x1049
>>> octets = b''.join(ndef.message_encoder([record]))
>>> len(octets)
139
```

### Connection Handover

Two NFC Devices in close proximity establish NFC communication based on the NFC Forum Logical Link Control Protocol (LLCP) specification. If one of the devices has intention to activate a further communication method, it can then use the NFC Forum Connection Handover protocol to announce possible communication means (potentially including configuration data) and request the other device to respond with a selection of matching technologies, including necessary configuration data.

An Enrollee NFC Device that has established NFC LLCP communication with a Registrar NFC Device sends a Connection Handover Request Message indicating Wi-Fi communication capability. A Registrar NFC Device responds with a Connection Handover Select Message indicating the Wi-Fi carrier which the Enrollee should associate with. The Enrollee is then provisioned by the Registrar through in-band WSC protocol message exchange (with encrypted ConfigData from the Registrar included in M2).

The following table shows the format of the Wi-Fi Carrier Configuration Record as transmitted within a Connection Handover Request Message. The UUID-E attribute is included to assist with the discovery over 802.11 that follows the exchange of the connection handover messages.

| Attribute | | Required/Conditional/Optional \| Description | |
|---|---|---|---|
| OOB Device Password | | R | A TLV with fixed data structure[1] |
| | Public Key Hash | R | The Enrollee's public key hash[2] |
| | Password ID | R | Set to NFC-Connection-Handover (0x0007) |
| UUID-E | | R | Universally Unique Identifier of the Enrollee Device |
| WFA Vendor Extension | | R | Vendor Extension with Vendor ID 00:37:2A[5] |
| | Version2 | R | Wi-Fi Simple Configuration version[6] |
| | <other . . . > | O | Other WFA Vendor Extension subelements |
| <other . . . > | | O | Other Wi-Fi Simple Configuration TLVs |

**Example:**

```
>>> import ndef
>>> import random
>>> import hashlib
>>> pkhash = hashlib.sha256(b'enrollee public key').digest()[0:20]
>>> oobpwd = ndef.wifi.OutOfBandPassword(pkhash, 0x0007, b'')
>>> wfaext = ndef.wifi.WifiAllianceVendorExtension(('version-2', b'\x20'))
>>> carrier = ndef.WifiSimpleConfigRecord()
>>> carrier.name = '0'
>>> carrier.set_attribute('oob-password', oobpwd)
>>> carrier.set_attribute('uuid-enrollee', '00010203-0405-0607-0809-0a0b0c0d0e0f')
>>> carrier['vendor-extension'] = [wfaext.encode()]
>>> print(carrier)
NDEF Wifi Simple Config Record ID '0' Attributes 0x102C 0x1047 0x1049
```

(continues on next page)

(continued from previous page)

```
>>> hr = ndef.handover.HandoverRequestRecord('1.3', random.randint(0, 0xffff))
>>> hr.add_alternative_carrier('active', carrier.name)
>>> octets = b''.join(ndef.message_encoder([hr, carrier]))
>>> len(octets)
108
```

The Wi-Fi Carrier Configuration Record transmitted within a Connection Handover Select Message from Registrar to Enrollee is shown below. The SSID attribute is included to assist with the discovery over 802.11 that follows the exchange of the connection handover messages. Optionally the RF Bands attribute, the AP Channel attribute and the MAC Address attribute may be included as hints to help the Enrollee find the AP without a full scan.

| Attribute | | Required/Conditional/Optional | Description |
|---|---|---|---|
| OOB Device Password | | R | A TLV with fixed data structure[1] |
| | Public Key Hash | R | The Registrar's public key hash[2] |
| | Password ID | R | Set to NFC-Connection-Handover (0x0007) |
| SSID | | R | Service Set Identifier of the network to connect |
| RF Bands | | O | Provides the operating RF band of the AP |
| AP Channel | | O | Provides the operating channel of the AP |
| MAC Address | | O | Basic Service Set Identifier of the AP |
| WFA Vendor Extension | | R | Vendor Extension with Vendor ID 00:37:2A[5] |
| | Version2 | R | Wi-Fi Simple Configuration version[6] |
| | <other . . . > | O | Other WFA Vendor Extension subelements |
| <other . . . > | | O | Other Wi-Fi Simple Configuration TLVs |

**Example:**

```
>>> import ndef
>>> import hashlib
>>> pkhash = hashlib.sha256(b'registrar public key').digest()[0:20]
>>> oobpwd = ndef.wifi.OutOfBandPassword(pkhash, 0x0007, b'')
>>> wfaext = ndef.wifi.WifiAllianceVendorExtension(('version-2', b'\x20'))
>>> carrier = ndef.WifiSimpleConfigRecord()
>>> carrier.name = '0'
>>> carrier.set_attribute('oob-password', oobpwd)
>>> carrier.set_attribute('ssid', b'802.11 network')
>>> carrier.set_attribute('rf-bands', '2.4GHz')
>>> carrier.set_attribute('ap-channel', 6)
>>> carrier.set_attribute('mac-address', b'\1\2\3\4\5\6')
>>> carrier['vendor-extension'] = [wfaext.encode()]
>>> print(carrier)
NDEF Wifi Simple Config Record ID '0' Attributes 0x1001 0x1020 0x102C 0x103C 0x1045␣
↪0x1049
>>> hs = ndef.handover.HandoverSelectRecord('1.3')
>>> hs.add_alternative_carrier('active', carrier.name)
>>> octets = b''.join(ndef.message_encoder([hs, carrier]))
>>> len(octets)
120
```

### 2.7.2 NDEF Record Classes

### Wi-Fi Simple Config Record

A *WifiSimpleConfigRecord* holds any number of Wi-Fi TLV (Type-Length-Value) Attributes which are defined in the Wi-Fi Simple Configuration specification. It is organized as a `dict` with numeric Attribute ID or symbolic *attribute_names* keys. Values are returned and must be set as a `list` of `bytes`, where each `bytes` object corresponds to one instance of the Wi-Fi TLV Attribute.

```
>>> import ndef
>>> record = ndef.WifiSimpleConfigRecord()
>>> record[0x1020] = [b'\x00\x01\x02\x03\x04\x05']
>>> assert record[0x1020] == record['mac-address']
>>> record['mac-address'].append(b'\x05\x04\x03\x02\x01\x00')
>>> record['mac-address']
[b'\x00\x01\x02\x03\x04\x05', b'\x05\x04\x03\x02\x01\x00']
```

The *get_attribute*, *set_attribute* and *add_attribute* methods can be used to get or set values using *WSC Attribute Classes*.

**class WifiSimpleConfigRecord**(**args*)

The *WifiSimpleConfigRecord* is initialized with any number of Wi-Fi Simple Config Attribute Type and Value tuples. The same Attribute Type may appear more than once.

```
>>> import ndef
>>> print(ndef.WifiSimpleConfigRecord((0x1001, b'\x00\x06'), ('ap-channel', b
↪'\x00\x06')))
NDEF Wifi Simple Config Record ID '' Attributes 0x1001 0x1001
```

**type**

The read-only Wifi Simple Configuration Record type.

```
>>> ndef.wifi.WifiSimpleConfigRecord().type
'application/vnd.wfa.wsc'
```

**name**

Value of the NDEF Record ID field, an empty `str` if not set.

```
>>> record = ndef.wifi.WifiSimpleConfigRecord()
>>> record.name = 'WSC Record'
>>> record.name
'WSC Record'
```

**data**

A `bytes` object containing the NDEF Record PAYLOAD encoded from the current attribute data.

```
>>> record = ndef.wifi.WifiSimpleConfigRecord()
>>> record.data
b''
>>> record['ap-channel'] = [b'\x00\x06']
>>> record.data
b'\x10\x01\x00\x02\x00\x06'
```

**attribute_names**

The read-only `list` of all WSC Attribute names that can be used as keys on the record instance or as names for the get/set/add_attribute methods.

```
>>> print('\n'.join(sorted(ndef.wifi.WifiSimpleConfigRecord().attribute_
↪names)))
ap-channel
credential
device-name
mac-address
manufacturer
model-name
model-number
oob-password
primary-device-type
rf-bands
secondary-device-type-list
serial-number
ssid
uuid-enrollee
uuid-registrar
vendor-extension
version-1
```

**get_attribute**(*name*, *index=0*)
The *get_attribute* method returns the Wi-Fi Attribute selected by *name* and *index*.

```
>>> record = ndef.WifiSimpleConfigRecord(('ap-channel', b'\x00\x06'))
>>> print(record.get_attribute('ap-channel', 0))
AP Channel 6
>>> print(record.get_attribute('ap-channel', 1))
None
```

**set_attribute**(*name*, **args*)
The *set_attribute* method sets the Wi-Fi Attribute *name* to a single instance constructed from *args*.

```
>>> record = ndef.WifiSimpleConfigRecord(('ap-channel', b'\x00\x06'))
>>> record.set_attribute('ap-channel', 10)
>>> print(record.get_attribute('ap-channel', 0))
AP Channel 10
>>> print(record.get_attribute('ap-channel', 1))
None
```

**add_attribute**(*name*, **args*)
The *add_attribute* method adds a Wi-Fi Attribute *name* constructed from *args* to any existing Wi-Fi Attributes of *name*. If there are no existing attributes for *name* the result is the same as for *set_attribute*.

```
>>> record = ndef.WifiSimpleConfigRecord(('ap-channel', b'\x00\x06'))
>>> record.add_attribute('ap-channel', 12)
>>> print(record.get_attribute('ap-channel', 0))
AP Channel 6
>>> print(record.get_attribute('ap-channel', 1))
AP Channel 12
```

### Wi-Fi Peer To Peer Record

**class WifiPeerToPeerRecord**(**args*)
The *WifiPeerToPeerRecord* inherits from *WifiSimpleConfigRecord* and adds handling of Wi-Fi P2P Attributes.

```
>>> import ndef
>>> print(ndef.WifiPeerToPeerRecord(('negotiation-channel', b'de\x04\x51\x06\x01
↪')))
NDEF Wifi Peer To Peer Record ID '' Attributes 0x13
```

**type**
The read-only Wifi Peer To Peer Record type.

```
>>> ndef.wifi.WifiPeerToPeerRecord().type
'application/vnd.wfa.p2p'
```

**attribute_names**
The read-only `list` of all WSC and P2P Attribute names that may be used as keys on the record instance or as names for the get/set/add_attribute methods.

```
>>> print('\n'.join(sorted(ndef.wifi.WifiPeerToPeerRecord().attribute_names)))
ap-channel
channel-list
credential
device-name
mac-address
manufacturer
model-name
model-number
negotiation-channel
oob-password
p2p-capability
p2p-device-info
p2p-group-id
p2p-group-info
primary-device-type
rf-bands
secondary-device-type-list
serial-number
ssid
uuid-enrollee
uuid-registrar
vendor-extension
version-1
```

### 2.7.3 WSC Attribute Classes

This section documents the Wi-Fi Simple Configuration (WSC) Attribute classes.

#### AP Channel

The AP Channel Attribute specifies the 802.11 channel that the AP is using.

**class** ndef.wifi.**APChannel**(*value*)
The *value* argument is the `int` or decimal integer `str` channel number.

```
>>> import ndef
>>> assert ndef.wifi.APChannel(6) == ndef.wifi.APChannel("6")
>>> ndef.wifi.APChannel(6).value
6
```

**value**
The read-only AP Channel `int` value.

### Authentication Type

The Authentication Type Attribute contains the authentication type for the Enrollee to use when associating with the network. For protocol version 2.0 or newer, the value 0x0022 can be used to indicate mixed mode operation (both WPA-Personal and WPA2-Personal enabled). All other values are required to have only a single bit set to one in this attribute value.

| Value | Authentication Type | Notes |
|---|---|---|
| 0x0001 | Open | |
| 0x0002 | WPA-Personal | deprecated in version 2.0 |
| 0x0004 | Shared | deprecated in version 2.0 |
| 0x0008 | WPA-Enterprise | deprecated in version 2.0 |
| 0x0010 | WPA2-Enterprise | includes both CCMP and GCMP |
| 0x0020 | WPA2-Personal | includes both CCMP and GCMP |

**class** ndef.wifi.**AuthenticationType**(**args*)
The *args* arguments may be a single `int` value with a bitwise OR of values from the authentication type table or one or more authentication type names. A type name can be used to test if the corresponding bit is set.

```
>>> import ndef
>>> mixed_mode = ndef.wifi.AuthenticationType('WPA-Personal', 'WPA2-Personal')
>>> mixed_mode.value
(34, 'WPA-Personal', 'WPA2-Personal')
>>> "WPA2-Personal" in mixed_mode
True
```

**value**
A tuple with the authentication type value and corresponding names.

### Configuration Methods

The Configuration Methods Attribute lists the configuration methods the Enrollee or Registrar supports.

| Value | Configuration Method | Description |
|---|---|---|
| 0x0001 | USBA | Deprecated |
| 0x0002 | Ethernet | Deprecated |
| 0x0004 | Label | 8 digit static PIN typically available on device. |
| 0x0008 | Display | A dynamic 4 or 8 digit PIN is available from a display.[10] |
| 0x0010 | External NFC Token | An NFC Tag transfers the configuration or device password. |
| 0x0020 | Integrated NFC Token | The NFC Tag is integrated in the device. |
| 0x0040 | NFC Interface | The device contains an NFC interface. |
| 0x0080 | PushButton | The device contains a physical or virtual pushbutton.[10] |
| 0x0100 | Keypad | Device is capable of entering a PIN |
| 0x0280 | Virtual Push Button | A virtual push button is avilable on a user interface. |
| 0x0480 | Physical Push Button | A physical push button is available on the device. |
| 0x2008 | Virtual Display PIN | The PIN is displayed through a remote user interface. |
| 0x4008 | Physical Display PIN | The PIN is shown on a display that is part of the device. |

**class** ndef.wifi.**ConfigMethods**(**args*)

The *args* arguments may be a single `int` value with a bitwise OR of values from the configuration method table or one or more method names. Any of the configuration method names can be tested for containment.

```
>>> import ndef
>>> config_methods = ndef.wifi.ConfigMethods("Label", "Display")
>>> assert ndef.wifi.ConfigMethods(0x000C) == config_methods
>>> "Label" in config_methods
True
>>> config_methods.value
(12, 'Label', 'Display')
```

**value**

A tuple with the configuration methods value and corresponding names.

### Credential

**class** ndef.wifi.**Credential**(**args*)

Credential is a compound Wi-Fi Attribute. It can be initialized with any number of Wi-Fi Attribute Type and Value tuples.

```
>>> import ndef
>>> credential = ndef.wifi.Credential(('ssid', b'my-ssid'), ('network-key', b
↪'secret'))
>>> print(credential)
Credential Attributes 0x1027 0x1045
>>> print(credential.get_attribute('ssid'))
SSID 6D:79:2D:73:73:69:64
```

**attribute_names**

A read-only `list` of all Wi-Fi Simple Configuration Attribute names that can be used as Credential keys.

```
>>> print('\n'.join(sorted(ndef.wifi.Credential().attribute_names)))
authentication-type
encryption-type
key-provided-automatically
mac-address
network-index
network-key
ssid
vendor-extension
```

**get_attribute**(*name*, *index=0*)

```
>>> import ndef
>>> credential = ndef.wifi.Credential(('mac-address', b'123456'))
>>> print(credential.get_attribute('mac-address'))
MAC Address 31:32:33:34:35:36
>>> print(credential.get_attribute('mac-address', 1))
None
```

**set_attribute**(*name*, **args*)

[10] Version 2.0 devices qualify a display as *Virtual Display PIN* or *Physical Display PIN* and a push button as *Virtual Push Button* or *Physical Push Button*.

```
>>> import ndef
>>> credential = ndef.wifi.Credential(('mac-address', b'123456'))
>>> credential.set_attribute('mac-address', b'654321')
>>> print(credential.get_attribute('mac-address'))
MAC Address 36:35:34:33:32:31
>>> print(credential.get_attribute('mac-address', 1))
None
```

**add_attribute**(*name*, **args*)

```
>>> import ndef
>>> credential = ndef.wifi.Credential(('mac-address', b'123456'))
>>> credential.add_attribute('mac-address', b'654321')
>>> print(credential.get_attribute('mac-address'))
MAC Address 31:32:33:34:35:36
>>> print(credential.get_attribute('mac-address', 1))
MAC Address 36:35:34:33:32:31
```

### Device Name

The Device Name Attribute contains a user-friendly description of the device encoded in UTF-8. Typically, this is a unique identifier that describes the product in a way that is recognizable to the user.

**class** ndef.wifi.**DeviceName**(*device_name*)

The *device_name* argument is unicode string of up to 32 characters.

**value**

The device name string.

### Encryption Type

The Encryption Type Attribute contains the encryption type for the Enrollee to use when associating with the network. For protocol version 2.0 or newer, the value 0x000C can be used to indicate mixed mode operation (both WPA-Personal with TKIP and WPA2-Personal with AES enabled). All other values are required to have only a single bit set to one in this attribute value.

| Value | Encryption Type | Notes |
|---|---|---|
| 0x0001 | None | |
| 0x0002 | WEP | Deprecated. |
| 0x0004 | TKIP | Deprecated. Use only for mixed mode. |
| 0x0008 | AES | Includes both CCMP and GCMP |

**class** ndef.wifi.**EncryptionType**(**args*)

The arguments *args* may be a single `int` value with a bitwise OR of values from the encryption type table or one or more encryption type names. A name can be used to test if that encryption type is included.

```
>>> import ndef
>>> mixed_mode = ndef.wifi.EncryptionType('TKIP', 'AES')
>>> assert ndef.wifi.EncryptionType(0x000C) == mixed_mode
>>> "AES" in mixed_mode
True
>>> mixed_mode.value
(12, 'TKIP', 'AES')
```

**value**
A tuple with the encryption type value and corresponding names.

### Key Provided Automatically

The Key Provided Automatically Attribute specifies whether the Network Key is provided automatically by the network.

**class** ndef.wifi.**KeyProvidedAutomatically**(*value*)
The *value* argument may be any type that can be converted into `bool`.

```
>>> import ndef
>>> ndef.wifi.KeyProvidedAutomatically(1).value
True
```

**value**
Either True or False.

### MAC Address

The MAC Address Attribute contains the 48 bit value of the MAC Address.

**class** ndef.wifi.**MacAddress**(*value*)
The *value* argument may be any type that can be converted to a `bytes` object with the six MAC Address octets.

```
>>> import ndef
>>> mac_address = ndef.wifi.MacAddress(b"\x01\x02\x03\x04\x05\x06")
>>> assert ndef.wifi.MacAddress([1, 2, 3, 4, 5, 6]) == mac_address
>>> mac_address.value
b'\x01\x02\x03\x04\x05\x06'
```

**value**
The six MAC Address bytes.

### Manufacturer

The Manufacturer Attribute is an ASCII string that identifies the manufacturer of the device. Generally, this should allow a user to make an association with the labeling on the device.

**class** ndef.wifi.**Manufacturer**(*value*)
The *value* argument is a text `str` or `bytes` containing ASCII characters.

```
>>> import ndef
>>> ndef.wifi.Manufacturer("Company").value
'Company'
```

**value**
The Manufacturer name string.

### Model Name

The Model Name Attribute is an ASCII string that identifies the model of the device. Generally, this field should allow a user to make an association with the labeling on the device.

**class** ndef.wifi.**ModelName**(*value*)
The *value* argument is a text `str` or `bytes` containing ASCII characters.

```
>>> import ndef
>>> ndef.wifi.ModelName("Product").value
'Product'
```

**value**
The Model Name string.

### Model Number

The Model Number Attribute provides additional description of the device to the user.

**class** ndef.wifi.**ModelNumber**(*value*)
The *value* argument is a text `str` or `bytes` containing ASCII characters.

```
>>> import ndef
>>> ndef.wifi.ModelNumber("007").value
'007'
```

**value**
The Model Number string.

### Network Index

The Network Index Attribute is deprecated. Value 1 must be used for backwards compatibility when the attribute is required.

**class** ndef.wifi.**NetworkIndex**(*value*)
The *value* argument is the `int` network index number.

```
>>> import ndef
>>> ndef.wifi.NetworkIndex(1).value
1
```

**value**
The Network Index integer.

### Network Key

The Network Key Attribute specifies the wireless encryption key to be used by the Enrollee.

**class** ndef.wifi.**NetworkKey**(*value*)
The *value* argument may be any type that can be converted to a `bytes` object with the 0 to 64 network key octets.

```
>>> import ndef
>>> ndef.wifi.NetworkKey(b"key").value
b'key'
```

**value**
The Network Key bytes.

### Network Key Shareable

The Network Key Shareable Attribute is used within Credential Attributes. It specifies whether the Network Key included in the Credential can be shared or not with other devices. A True value indicates that the Network Key can be shared.

**class** ndef.wifi.**NetworkKeyShareable**(*value*)

The *value* argument may be any type that can be converted into `bool`.

```
>>> import ndef
>>> ndef.wifi.NetworkKeyShareable(True).value
True
```

**value**

Either True or False.

### Out Of Band Device Password

The Out-of-Band Device Password Attribute contains a fixed data structure with the overall size is given by the Wi-Fi Attribute TLV Length value.

| Field | Size | Description |
|---|---|---|
| Public Key Hash | 20 | First 160 bits of the public key hash. |
| Password ID | 2 | 16 bit identifier for the device password. |
| Device Password | 16-32 | Zero or 16–32 octet long device password. |

The Password ID of an Out-of-Band Device Password must be between 0x0010 and 0xFFFF inclusively and chosen at random, except when NFC negotiated handover is used in which case the Password ID is set to 0x0007.

The Device Password is (Length – 22) octets long, with a maximum size of 32 octets. A Device Password length of 32 byte is recommended if the out-of-band channel has sufficient capacity. Otherwise, it can be any size with a minimum length of 16 bytes, except when the Password ID is equal to 0x0007 (NFC negotiated handover) in which case it has zero length.

For Enrollee provided Device Passwords, the Public Key Hash Data field corresponds to the first 160 bits of a SHA-256 hash of the Enrollee's public key exchanged in message M1. For Registrar provided Device Passwords, the Public Key Hash Data field corresponds to the first 160 bits of a SHA-256 hash of the Registrar's public key exchanged in message M2.

**class** ndef.wifi.**OutOfBandPassword**(*public_key_hash*, *password_id*, *password*)

The *public_key_hash* attribute is a `bytes` object with the first 20 octets of the SHA-256 hash of the device's public key. The *password_id* argument is a 16-bit unsigned `int` value. The *password* is a `bytes` object with the either 0 or 16-32 octets long device password.

```
>>> import ndef
>>> import random
>>> import hashlib
>>> pubkey_hash = hashlib.sha256(b'my public key goes here').digest()[0:20]
>>> password_id = random.randint(16, 65535)
>>> my_password = b"my long password you can't guess"
>>> oob = ndef.wifi.OutOfBandPassword(pubkey_hash, password_id, my_password)
>>> assert oob.value == (pubkey_hash, password_id, my_password)
>>> assert oob.public_key_hash == pubkey_hash
>>> assert oob.password_id == password_id
>>> assert oob.device_password == b"my long password you can't guess"
```

**value**
The Out Of Band Password Attribute as the (public_key_hash, password_id, password).

**public_key_hash**
The Public Key Hash bytes.

**password_id**
The Password ID integer.

**device_password**
The Device Password bytes.

### Primary Device Type

The Primary Device Type Attribute contains the primary type of the device.

```
"Computer::PC"
"Computer::Server"
"Computer::MediaCenter"
"Computer::UltraMobile"
"Computer::Notebook"
"Computer::Desktop"
"Computer::MobileInternetDevice"
"Computer::Netbook"
"Computer::Tablet"
"Computer::Ultrabook"
"Input::Keyboard"
"Input::Mouse"
"Input::Joystick"
"Input::Trackball"
"Input::GameController"
"Input::Remote"
"Input::Touchscreen"
"Input::BiometricReader"
"Input::BarcodeReader"
"Printer::Scanner"
"Printer::Fax"
"Printer::Copier"
"Printer::Multifunction"
"Camera::DigitalStillCamera"
"Camera::VideoCamera"
"Camera::WebCamera"
"Camera::SecurityCamera"
"Storage::NAS"
"Network::AccessPoint"
"Network::Router"
"Network::Switch"
"Network::Gateway"
"Network::Bridge"
"Display::Television"
"Display::PictureFrame"
"Display::Projector"
"Display::Monitor"
"Multimedia::DigitalAudioRecorder"
"Multimedia::PersonalVideoRecorder"
"Multimedia::MediaCenterExtender"
"Multimedia::SetTopBox"
"Multimedia::ServerAdapterExtender"
```

(continues on next page)

(continued from previous page)

```
"Multimedia::PortableVideoPlayer"
"Gaming::Xbox"
"Gaming::Xbox360"
"Gaming::Playstation"
"Gaming::Console"
"Gaming::Portable"
"Telephone::WindowsMobile"
"Telephone::SingleModePhone"
"Telephone::DualModePhone"
"Telephone::SingleModeSmartphone"
"Telephone::DualModeSmartphone"
"Audio::Receiver"
"Audio::Speaker"
"Audio::PortableMusicPlayer"
"Audio::Headset"
"Audio::Headphone"
"Audio::Microphone"
"Audio::HomeTheater"
"Dock::Computer"
"Dock::Media"
```

**class** ndef.wifi.**PrimaryDeviceType**(*value*)

The *value* attribute may be either a 64-bit integer equivalent to the Attribute Value bytes in MSB order, or one of the text values above.

```
>>> import ndef
>>> device_type_1 = ndef.wifi.PrimaryDeviceType(0x00010050F2040001)
>>> device_type_2 = ndef.wifi.PrimaryDeviceType("Computer::PC")
>>> assert device_type_1 == device_type_2
>>> device_type_1.value
'Computer::PC'
>>> ndef.wifi.PrimaryDeviceType(0x0001FFFFFF000001).value
'Computer::FFFFFF000001'
>>> ndef.wifi.PrimaryDeviceType(0xABCDFFFFFF000001).value
'ABCD::FFFFFF000001'
```

**value**

The Primary Device Type string.

### RF Bands

The RF Bands Attribute indicates a specific RF band that is utilized during message exchange. As an optional attribute in NFC out-of-band provisioning it indicates the RF Band relating to a channel or the RF Bands in which an AP is operating with a particular SSID.

| Value | RF Band |
|---|---|
| 0x01 | 2.4GHz |
| 0x02 | 5.0GHz |
| 0x03 | 60GHz |

**class** ndef.wifi.**RFBands**(**args*)

The arguments *args* may be a single `int` value with a bitwise OR of values from the RF bands table or one or more RF band names. A name can be used to test if that RF band is included.

```
>>> import ndef
>>> assert ndef.wifi.RFBands(0x03) == ndef.wifi.RFBands('2.4GHz', '5.0GHz')
>>> "5.0GHz" in ndef.wifi.RFBands(0x03)
True
>>> ndef.wifi.RFBands(0x03).value
(3, '2.4GHz', '5.0GHz')
```

**value**
The tuple of RF Bands integer value and corresponding names.

### Secondary Device Type List

The Secondary Device Type List contains one or more secondary device types supported by the device. The standard values of Category and Sub Category are the same as for the *Primary Device Type* Attribute.

**class SecondaryDeviceTypeList**(**args*)
One or more initialization arguments my be supplied as 64-bit integers or device type strings.

```
>>> import ndef
>>> ndef.wifi.SecondaryDeviceTypeList(0x00010050F2040002, 'Storage::NAS').value
('Computer::Server', 'Storage::NAS')
```

**value**
A tuple of all device type strings.

### Serial Number

The Serial Number Attribute contains the serial number of the device.

**class** ndef.wifi.**SerialNumber**(*value*)
The *value* argument is a text `str` or `bytes` containing ASCII characters.

```
>>> import ndef
>>> ndef.wifi.SerialNumber("CB5A281NNP").value
'CB5A281NNP'
```

**value**
The Serial Number string.

### SSID

The SSID Attribute represents the Service Set Identifier a.k.a network name. This is used by the client to identify the wireless network to connect with. The SSID Attribute value must match exactly with the value of the SSID, i.e. no zero padding and same length.

**class** ndef.wifi.**SSID**
The *value* argument may be any type that can be converted to a `bytes` object with the SSID octets.

```
>>> import ndef
>>> ndef.wifi.SSID(b"my wireless network").value
b'my wireless network'
```

**value**
The SSID bytes.

### UUID-E

The UUID-E Attribute contains the universally unique identifier (UUID) generated as a GUID by the Enrollee. It uniquely identifies an operational device and should survive reboots and resets.

**class** ndef.wifi.**UUIDEnrollee**(*value*)

The *value* argument may be either a `uuid.UUID` object, or the 16 `bytes` of a UUID, or any `str` value that can be used to initialize `uuid.UUID` object.

```
>>> import ndef
>>> ndef.wifi.UUIDEnrollee(bytes(range(16))).value
'00010203-0405-0607-0809-0a0b0c0d0e0f'
>>> ndef.wifi.UUIDEnrollee("00010203-0405-0607-0809-0a0b0c0d0e0f").value
'00010203-0405-0607-0809-0a0b0c0d0e0f'
```

**value**

The UUID-E string.

### UUID-R

The UUID-R Attribute contains the universally unique identifier (UUID) generated as a GUID by the Registrar. It uniquely identifies an operational device and should survive reboots and resets.

**class** ndef.wifi.**UUIDRegistrar**

The *value* argument may be either a `uuid.UUID` object, or the 16 `bytes` of a UUID, or any `str` value that can be used to initialize `uuid.UUID` object.

```
>>> import ndef
>>> ndef.wifi.UUIDRegistrar(bytes(range(16))).value
'00010203-0405-0607-0809-0a0b0c0d0e0f'
>>> ndef.wifi.UUIDRegistrar('00010203-0405-0607-0809-0a0b0c0d0e0f').value
'00010203-0405-0607-0809-0a0b0c0d0e0f'
```

**value**

The UUID-E string.

### Version

The Version Attribute is deprecated and always set to 0x10 (version 1.0) for backwards compatibility. Version 1.0h of the specification did not fully describe the version negotiation mechanism and version 2.0 introduced a new subelement (Version2) for indicating the version number to avoid potential interoperability issues with deployed 1.0h-based devices.

**class** ndef.wifi.**Version1**(**args*)

A single argument provides the version number as an 8-bit unsigned `int`. Two arguments provide the major and minor version numbers as 4-bi unsigned `int`.

```
>>> import ndef
>>> assert ndef.wifi.Version1(0x10) == ndef.wifi.Version1(1, 0)
>>> ndef.wifi.Version1(1, 0).value
Version(major=1, minor=0)
```

**value**

The Version as a `namedtuple` with major and minor fields.

### Version2

The Version2 Attribute specifies the Wi-Fi Simple Configuration version implemented by the device sending this attribute. It is a subelement within a Wi-Fi Alliance Vendor Extension that was added in the specification version 2.0. If the Version2 Attribute is not included in a message it is assumed to use version 1.0.

**class** ndef.wifi.**Version2**(**args*)

A single argument provides the version number as an 8-bit unsigned `int`. Two arguments provide the major and minor version numbers as 4-bit unsigned `int`.

```
>>> import ndef
>>> assert ndef.wifi.Version2(0x20) == ndef.wifi.Version2(2, 0)
>>> ndef.wifi.Version1(2, 0).value
Version(major=2, minor=0)
```

**value**

The Version2 as a `namedtuple` with major and minor fields.

### Vendor Extension

The Vendor Extension Attribute allows vendor specific extensions in the Wi-Fi Simple Configuration message formats. The Vendor Extension Value field contains the Vendor ID followed by a maximum of 1021 octets Vendor Data. Vendor ID is the SMI network management private enterprise code.

**class** ndef.wifi.**VendorExtension**(*vendor_id*, *vendor_data*)

Both the *vendor_id* and *vendor_data* arguments are `bytes` that initalize the fields to encode. The *vendor_id* must be 3 octets while *vendor_data* may contain from 0 to 1021 octets.

```
>>> import ndef
>>> vendor_id, vendor_data = (b'\x00\x37\x2A', b'123')
>>> ndef.wifi.VendorExtension(vendor_id, vendor_data).value == (vendor_id, vendor_
↪data)
True
```

**value**

The read-only Vendor Extension Attribute as the `tuple` of (vendor_id, vendor_data).

### Wi-Fi Alliance Vendor Extension

The Wi-Fi Alliance (WFA) Vendor Extension is a Vendor Extension attribute (ID 0x1049) that uses Vendor ID 0x00372A and contains one or more subelements. The WFA Vendor Extension attribute is used to encode new information in a way that avoids some backwards compatibility issues with deployed implementations that are based on previous specification versions, but do not comply with requirements to ignore new attributes.

**class** ndef.wifi.**WifiAllianceVendorExtension**

The *`WifiAllianceVendorExtension`* is an attribute container class that holds other Wi-Fi Simple Configuration attributes. It may be initialzed with any number of WFA sublement type-value tuples.

```
>>> import ndef
>>> wfa_ext = ndef.wifi.WifiAllianceVendorExtension(('version-2', b'\x20'))
>>> wfa_ext[0x02] = [b'\x01'] # network key shareable
>>> print(wfa_ext)
WFA Vendor Extension Attributes 0x00 0x02
```

**attribute_names**
The read-only list of all WSC attribute names (subelements) that may be used as a key or name for the get/set/add_attribute methods.

```
>>> print('\n'.join(sorted(ndef.wifi.WifiAllianceVendorExtension().attribute_
↪names)))
network-key-shareable
version-2
```

**get_attribute**(*name*, *index=0*)
The *get_attribute* method returns the WFA subelement attribute selected by name and index.

```
>>> wfa_ext = ndef.wifi.WifiAllianceVendorExtension(('version-2', b'\x20'))
>>> wfa_ext.get_attribute('version-2')
ndef.wifi.Version2(2, 0)
```

**set_attribute**(*name*, **args*)
The *set_attribute* method sets the WFA subelement attribute *name* to a single instance constructed from *args*.

```
>>> wfa_ext = ndef.wifi.WifiAllianceVendorExtension(('version-2', b'\x20'))
>>> wfa_ext.set_attribute('version-2', 0x21)
>>> wfa_ext.get_attribute('version-2')
ndef.wifi.Version2(2, 1)
```

**add_attribute**(*name*, **args*)
The *add_attribute* method adds a WFA subelement attribute *name* constructed from *args* to any existing *name* attributes. If there are no existing *name* attributes it is effectively the same as *set_attribute*.

```
>>> wfa_ext = ndef.wifi.WifiAllianceVendorExtension()
>>> wfa_ext.add_attribute('version-2', ndef.wifi.Version2(2, 0))
>>> wfa_ext.add_attribute('version-2', ndef.wifi.Version2(2, 1))
>>> wfa_ext.get_attribute('version-2', 0)
ndef.wifi.Version2(2, 0)
>>> wfa_ext.get_attribute('version-2', 1)
ndef.wifi.Version2(2, 1)
```

### 2.7.4 P2P Attribute Classes

This section documents the Wi-Fi Peer To Peer (P2P) Attribute classes.

#### P2P Capability

The P2P Capability attribute contains a set of parameters that indicate the P2P Device's capability and the current state of the P2P Group.

Device Capability Strings:

```
'Service Discovery'
'P2P Client Discoverability'
'Concurrent Operation'
'P2P Infastructure Managed'
'P2P Device Limit'
'P2P Invitation Procedure'
```

(continues on next page)

(continued from previous page)

```
'Reserved Bit 6'
'Reserved Bit 7'
```

Group Capability Strings:

```
'P2P Group Owner'
'Persistent P2P Group'
'P2P Group Limit'
'Intra-BSS Distribution'
'Cross Connection'
'Persistent Reconnect'
'Group Formation'
'IP Address Allocation'
```

**class** ndef.wifi.**PeerToPeerCapability**(*device_capability*, *group_capability*)

Both init arguments *device_capability* and *group_capability* may be set as either 8-bit integer values with each bit position corresponding to an individual capability, or as a list of capability strings.

```
>>> import ndef
>>> attr_1 = ndef.wifi.PeerToPeerCapability(0b00000001, 0b01000000)
>>> attr_2 = ndef.wifi.PeerToPeerCapability(['Service Discovery'], ['Group␣
↪Formation'])
>>> assert attr_1 == attr_2
>>> ndef.wifi.PeerToPeerCapability(3, 65).device_capability
(3, 'Service Discovery', 'P2P Client Discoverability')
```

**device_capability**

The P2P Device Capabilities as a tuple with the first element the numerical value of the device capability bitmap and following elements are capability strings. This attribute is read-only.

```
>>> import ndef
>>> ndef.wifi.PeerToPeerCapability(3, 0).device_capability
(3, 'Service Discovery', 'P2P Client Discoverability')
```

**group_capability**

The P2P Group Capabilities as a tuple with the first element the numerical value of the group capability bitmap and following elements are capability strings. This attribute is read-only.

```
>>> import ndef
>>> ndef.wifi.PeerToPeerCapability(0, 65).group_capability
(65, 'P2P Group Owner', 'Group Formation')
```

### Channel List

The Channel List attribute contains a list of Operating Class and Channel pair information.

**class** ndef.wifi.**ChannelList**(*country_string*, **channel_entry*)

The *country_string* argument determines the country code for the *channel_entry* argument(s). Each *channel_entry* is a tuple of an *operating_class* integer and a *channel_numbers* list.

```
>>> import ndef
>>> channel_list = ndef.wifi.ChannelList(b"de\x04", (81, (1, 6)), (115, (36, 44)))
>>> print(channel_list)
Channel List Country DE Table E-4 Class 81 Channels [1, 6], Class 115 Channels␣
↪[36, 44]
```

(continues on next page)

(continued from previous page)

```
>>> len(channel_list)
2
>>> print(channel_list[0])
Class 81 Channels [1, 6]
>>> channel_list[0].operating_class
81
>>> channel_list[0].channel_numbers
(1, 6)
```

**country_string**
The Country String field is the value contained in the dot11CountryString attribute, specifying the country code in which the Channel Entry List is valid. The third octet of the Country String field is always set to hex 04 to indicate that Table E-4 is used.

```
>>> import ndef
>>> ndef.wifi.ChannelList(b"de\x04", (81, (1,))).country_string
b'de\x04'
```

### P2P Device Info

The P2P Device Info attribute provides the P2P Device Address, Config Methods, Primary Device Type, a list of Secondary Device Types and the user friendly Device Name.

**class** ndef.wifi.**PeerToPeerDeviceInfo**(*adr*, *cfg*, *pdt*, *sdtl*, *name*)
The first argument *adr* must be the 6 bytes P2P Device Address. The *cfg* argument is a tuple of *Configuration Methods* strings. The *pdt* argument specifies the *Primary Device Type* of the P2P Device as a single text string. The *Secondary Device Type List* *sdtl* argument expects a tuple of device type strings. The *Device Name* *name* argument provides the friendly name of the P2P Device. All arguments must be supplied.

```
>>> import ndef
>>> adr = b'\x01\x02\x03\x04\x05\x06'
>>> cfg = ('Label', 'Display')
>>> pdt = 'Computer::Tablet'
>>> sdtl = ('Computer::PC', )
>>> name = 'my tablet'
>>> info = ndef.wifi.PeerToPeerDeviceInfo(adr, cfg, pdt, sdtl, name)
>>> print(info)
P2P Device Info 01:02:03:04:05:06 0x000C ['Label', 'Display'] Computer::Tablet␣
↪Computer::PC 'my tablet'
```

**device_address**
The P2P Device Identifier used to uniquely reference a P2P Device returned as a 6 byte string. The *device_address* attribute is read-only.

```
>>> info.device_address
b'\x01\x02\x03\x04\x05\x06'
```

**config_methods**
The *Configuration Methods* that are supported by this device e.g. PIN from a Keypad, PBC etc. The values are returned as a tuple where the first entry is the config methods bitmap and remaining entries are method strings. The *config_methods* attribute is read-only.

```
>>> info.config_methods
(12, 'Label', 'Display')
```

**primary_device_type**
The Primary Device Type of the P2P Device returned as a string. See *Primary Device Type* for representation of pre-defined and custom values. The *primary_device_type* attribute is read-only.

```
>>> info.primary_device_type
'Computer::Tablet'
```

**secondary_device_type_list**
A list of Secondary Device Types of the P2P Client. Returns a, potentially empty, tuple of device type strings. The *secondary_device_type_list* attribute is read-only.

```
>>> info.secondary_device_type_list
('Computer::PC',)
```

**device_name**
The friendly name of the P2P Device which should be the same as the WSC *Device Name*. The *device_name* attribute is read-only.

```
>>> info.device_name
'my tablet'
```

### P2P Group Info

The P2P Group Info attribute contains device information of P2P Clients that are members of the P2P Group.

**class** ndef.wifi.**PeerToPeerGroupInfo**(**client_info*)
A `PeerToPeerGroupInfo` object holds a number of client info descriptors. It is initialized with a number of client info data tuples as shown below.

```
>>> import ndef
>>> client_info_1 = (
...     b'\x01\x02\x03\x04\x05\x06',  # P2P Device Address
...     b'\x11\x12\x13\x14\x15\x16',  # P2P Interface Address
...     ('Service Discovery',),       # Device Capabilities
...     ('NFC Interface',),           # Configuration Methods
...     "Computer::Tablet",           # Primary Device Type
...     (),                           # Secondary Device Types
...     'first device',               # Device name
... )
>>> client_info_2 = (
...     b'\x21\x22\x23\x24\x25\x26',  # P2P Device Address
...     b'\x31\x32\x33\x34\x35\x36',  # P2P Interface Address
...     ('Service Discovery',),       # Device Capabilities
...     ('NFC Interface',),           # Configuration Methods
...     "Computer::Tablet",           # Primary Device Type
...     (),                           # Secondary Device Types
...     'second device',              # Device name
... )
>>> group_info = ndef.wifi.PeerToPeerGroupInfo(client_info_1, client_info_2)
>>> print(group_info)
P2P Group Info (Device 1: 01:02:03:04:05:06 11:12:13:14:15:16 Capability [
↪'Service Discovery'] Config 0x0040 ['NFC Interface'] Type 'Computer::Tablet '␣
↪Name 'first device'), (Device 2: 21:22:23:24:25:26 31:32:33:34:35:36 Capability␣
↪['Service Discovery'] Config 0x0040 ['NFC Interface'] Type 'Computer::Tablet '␣
↪Name 'second device')
>>> [client_info.device_name for client_info in group_info]
```

(continues on next page)

(continued from previous page)

```
['first device', 'second device']
>>> type(group_info[0])
<class 'ndef.wifi.PeerToPeerGroupInfo.Descriptor'>
```

**class Descriptor**

P2P Client Info within a `PeerToPeerGroupInfo` is exposed as a `Descriptor` instance with attributes for the relevant information fields.

```
>>> descriptor = group_info[0]
```

**device_address**

The 6 byte P2P Device Identifier used to uniquely reference a P2P Device. The *device_address* attribute is read-only.

```
>>> descriptor.device_address
b'\x01\x02\x03\x04\x05\x06'
```

**interface_address**

The 6 byte P2P Interface Address is used to identify a P2P Device within a P2P Group. The *interface_address* attribute is read-only.

```
>>> descriptor.interface_address
b'\x11\x12\x13\x14\x15\x16'
```

**config_methods**

The *Configuration Methods* that are supported by this device e.g. PIN from a Keypad, PBC etc. The values are returned as a tuple where the first entry is the config methods bitmap and remaining entries are method strings. The *config_methods* attribute is read-only.

```
>>> descriptor.config_methods
(64, 'NFC Interface')
```

**primary_device_type**

The Primary Device Type of the P2P Device returned as a string. See *Primary Device Type* for representation of pre-defined and custom values. The *primary_device_type* attribute is read-only.

```
>>> descriptor.primary_device_type
'Computer::Tablet'
```

**secondary_device_type_list**

A list of Secondary Device Types of the P2P Client. Returns a, potentially empty, tuple of device type strings. The *secondary_device_type_list* attribute is read-only.

```
>>> descriptor.secondary_device_type_list
()
```

**device_name**

The friendly name of the P2P Device which should be the same as the WSC *Device Name*. The *device_name* attribute is read-only.

```
>>> descriptor.device_name
'first device'
```

### P2P Group ID

The P2P Group ID attribute contains a unique P2P Group identifier of the P2P Group.

**class** ndef.wifi.**PeerToPeerGroupID**(*device_address*, *ssid*)

Both the *device_address* and *ssid* arguments must be given as byte strings and the *device_address* must be exactly 6 byte long.

```
>>> import ndef
>>> attr = ndef.wifi.PeerToPeerGroupID(b'\1\2\3\4\5\6', b'P2P Group SSID')
>>> print(attr)
P2P Group ID 01:02:03:04:05:06 SSID 50:32:50:20:47:72:6F:75:70:20:53:53:49:44
```

**device_address**

The 6 byte P2P Device Identifier used to uniquely reference a P2P Device. The *device_address* attribute is read-only.

```
>>> attr.device_address
b'\x01\x02\x03\x04\x05\x06'
```

**ssid**

The service set identifier (a.k.a. network name) as a byte string. Although often printable it is in fact just a sequence of bytes with no implied text encoding. The *ssid* attribute is read-only.

```
>>> attr.ssid
b'P2P Group SSID'
```

### Negotiation Channel

The Out-of-Band Group Owner Negotiation Channel attribute contains the Channel and Class information used for the Group Owner Negotiation.

**class** ndef.wifi.**NegotiationChannel**(*country_string*, *operating_class*, *channel_number*, *role_indication*)

The *country_string* argument specifies the country code and operating class table (always value 0x04) in 3 bytes. The *operating_class* and *channel_number* must be 8-bit integer values. The *role_indication* argument must be either `'Not Member'`, `'Group Client'`, or `'Group Owner'`.

```
>>> import ndef
>>> attr = ndef.wifi.NegotiationChannel(b'de\x04', 81, 6, 'Group Client')
>>> print(attr)
Negotiation Channel Country DE Table E-4 Class 81 Channel 6 Role 'Group Client'
```

**country_string**

The Country String specifies the country code in which the Group Formation Class and Channel Number fields are valid. The third octet of the Country String is set to hex 04 to indicate that Table E-4 is used. The *country_string* attribute is read-only.

```
>>> attr.country_string
b'de\x04'
```

**operating_class**

Provides the preferred Operating Class for the Group Owner Negotiation. An Operating Class value 0 indicates that no preferred Operating Class is available. If set to 0, the Operating Class information provided in the Channel List attribute shall be used.

```
>>> attr.operating_class
81
```

**channel_number**
Provides the preferred channel for the Group Formation. A Channel Number value 0 indicates that no group formation preferred channel is available and P2P Group Owner negotiation with a full channel search based on the information provided in the Channel List attribute shall be used.

```
>>> attr.channel_number
6
```

**role_indication**
Indicates the current role of the P2P device. It reads as a 2-tuple where the first value is the numerical and the second value the textual representation. The *role_indication* attribute is read-only.

```
>>> attr.role_indication
(1, 'Group Client')
```

## 2.8 Signature Record

The NDEF Signature Record is a well-known record type defined by the NFC Forum. It contains three fields: a version field, a signature field and a certificate field.

The version field is static. Currently this implementation only supports v2.0 of the NDEF Signature Record.

The signature field contains the signature type, the message hash type and either the signature itself or a URI to the signature.

The certificate field contains the certificate format type, a certificate chain store and an option URI to the next certificate in the chain.

**class SignatureRecord**(*signature_type=None*, *hash_type='SHA-256'*, *signature=b''*, *signature_uri=''*, *certificate_format='X.509'*, *certificate_store=[]*, *certificate_uri=''*)
The *SignatureRecord* class decodes or encodes an NDEF Signature Record.

**Parameters**

- **signature_type** (*str*) – initial value for the *signature_type* attribute, default None
- **hash_type** (*str*) – initial value for the *hash_type* attribute, default 'SHA-256'
- **signature** (*bytes*) – initial value for the *signature* attribute, default b''
- **signature_uri** (*str*) – initial value for the *signature_uri* attribute, default ''
- **certificate_format** (*str*) – initial value for the *certificate_format* attribute, default 'X.509'
- **certificate_store** (*list*) – initial value for the *certificate_store* attribute, default []
- **certificate_uri** (*str*) – initial value for the *certificate_uri* attribute, default ''

**type**
The Signature Record type is `urn:nfc:wkt:Sig`.

**name**
Value of the NDEF Record ID field, an empty `str` if not set.

**data**
A `bytes` object containing the NDEF Record PAYLOAD encoded from the current attributes.

**version**
The version of the NDEF Signature Record.

**signature_type**
The signature type used in the signature algorithm.

```
>>> import ndef
>>> print('\n'.join([str(x[1]) for x in ndef.signature.SignatureRecord()._
↪mapping_signature_type]))
None
RSASSA-PSS-1024
RSASSA-PKCS1-v1_5-1024
DSA-1024
ECDSA-P192
RSASSA-PSS-2048
RSASSA-PKCS1-v1_5-2048
DSA-2048
ECDSA-P224
ECDSA-K233
ECDSA-B233
ECDSA-P256
```

**hash_type**
The hash type used in the signature algorithm.

```
>>> import ndef
>>> print("\n".join([str(x[1]) for x in ndef.signature.SignatureRecord()._
↪mapping_hash_type]))
SHA-256
```

**signature**
The signature (if not specified by *signature_uri*).

**signature_uri**
The uniform resource identifier for the signature (if not specified by *signature*).

**certificate_format**
The format of the certificates in the chain.

```
>>> import ndef
>>> print("\n".join([str(x[1]) for x in ndef.signature.SignatureRecord()._
↪mapping_certificate_format]))
X.509
M2M
```

**certificate_store**
A list of certificates in the certificate chain.

**certificate_uri**
The uniform resource identifier for the next certificate in the certificate chain.

This is default usage:

```
>>> import ndef
>>> signature_record = ndef.SignatureRecord(None, 'SHA-256', b'', '', 'X.509', [],
↪ '')
```

This is a full example creating records, signing them and verifying them:

```
>>> import ndef
>>> import io
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import ec
>>> from cryptography.hazmat.primitives.asymmetric import utils
>>> from cryptography.exceptions import InvalidSignature
>>> from asn1crypto.algos import DSASignature
```

```
>>> private_key = ec.generate_private_key(ec.SECP256K1(), default_backend())
>>> public_key = private_key.public_key()
```

```
>>> r1 = ndef.UriRecord("https://example.com")
>>> r2 = ndef.TextRecord("TEST")
```

```
>>> stream = io.BytesIO()
>>> records = [r1, r2, ndef.SignatureRecord("ECDSA-P256", "SHA-256")]
>>> encoder = ndef.message_encoder(records, stream)
>>> for _ in range(len(records) - 1): e=next(encoder)
```

```
>>> signature = private_key.sign(stream.getvalue(), ec.ECDSA(hashes.SHA256()))
>>> records[-1].signature = DSASignature.load(signature, strict=True).to_p1363()
>>> e=next(encoder)
>>> octets = stream.getvalue()
```

```
>>> records_verified = []
>>> records_to_verify = []
>>> known_types = {'urn:nfc:wkt:Sig': ndef.signature.SignatureRecord}
>>> for record in ndef.message_decoder(octets, known_types=known_types):
...     if not record.type == 'urn:nfc:wkt:Sig':
...         records_to_verify.append(record)
...     else:
...         stream_to_verify = io.BytesIO()
...         encoder_to_verify = ndef.message_encoder(records_to_verify + [record],
↪ stream_to_verify)
...         for _ in range(len(records_to_verify)): e=next(encoder_to_verify)
...         try:
...             public_key.verify(DSASignature.from_p1363(record.signature).
↪dump(), stream_to_verify.getvalue(), ec.ECDSA(hashes.SHA256()))
...             records_verified.extend(records_to_verify)
...             records_to_verify = []
...         except InvalidSignature:
...             pass
```

```
>>> records_verified = list(ndef.message_decoder(b''.join(ndef.message_
↪encoder(records_verified))))
```

CHAPTER 3

# Adding Private Records

Private (or experimental) NDEF Record decoding and encoding can be easily made recognized by the *message_decoder()* and *message_encoder()*. It just requires a record class that inherits from ndef.record.GlobalRecord and provides the desired record type value as well as the payload decode and encode methods. The following sections document the decode/encode interface by way of example, with increasing complexity.

## 3.1 Record with no Payload

This is the most simple yet fully functional record class. It inherits from the abstract class ndef.record.GlobalRecord (which is actually just an abstract version of *Record* to make sure the dervied class implements the payload decode and encode methods. The record type string is set via the _type class attribute. The _encode_payload method must return the bytes for the NDEF Record PAYLOAD field, usually encoded from other record attributes but here it's just empty. The _decode_payload classmethod receives the NDEF Record PAYLOAD field the bytes type *octets* and returns a record object populated with the decoded PAYLOAD data, again nothing for the record with no payload. The _decode_min_payload_length and _decode_max_payload_length class attributes (put at the end of the class definition only to align with the explanation) inform the record decoder about the minmum required and maximum acceptable PAYLOAD size, thus the *octets* argument will never have less or more data. If a class does not set those values, the default min value is 0 and the default max value is *Record.MAX_PAYLOAD_SIZE*.

```
import ndef

class ExampleRecordWithNoPayload(ndef.record.GlobalRecord):
    """An NDEF Record with no payload."""

    _type = 'urn:nfc:ext:nfcpy.org:x-empty'

    def _encode_payload(self):
        # This record does not have any payload to encode.
        return b''
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def _decode_payload(cls, octets, errors):
        # This record does not have any payload to decode.
        return cls()

    _decode_min_payload_length = 0
    _decode_max_payload_length = 0

ndef.Record.register_type(ExampleRecordWithNoPayload)

record = ExampleRecordWithNoPayload()
octets = b''.join(ndef.message_encoder([record]))
print("encoded: {}".format(octets))

message = list(ndef.message_decoder(octets))
print("decoded: {}".format(message[0]))
```

```
encoded: b'\xd4\x11\x00nfcpy.org:x-empty'
decoded: NDEF Example Record With No Payload ID '' PAYLOAD 0 byte
```

## 3.2 Example Temperature Record

This record carries an unsigned 32-bit integer timestamp that is the seconds since 1.1.1970 (and will overflow on February 7, 2106 !) and a signed 16-bit integer with a temperature. The payload is thus a fixed structure with exactly 6 octets for which the inherited `_decode_struct` and `_encode_struct` methods are perfectly suited. They are quite the same as using `struct.unpack_from` and `struct.pack` but return a single value directly and not as a (value, ) tuple.

This example also shows how the `__format__` method is used to provide an arguments and a data view for the `str()` and `repr()` functions.

```
import ndef
import time

class ExampleTemperatureRecord(ndef.record.GlobalRecord):
    """An NDEF Record that carries a temperature and a timestamp."""

    _type = 'urn:nfc:ext:nfcpy.org:x-temp'

    def __init__(self, timestamp, temperature):
        self._time = timestamp
        self._temp = temperature

    def __format__(self, format_spec):
        if format_spec == 'args':
            # Return the init args for repr() but w/o class name and brackets
            return "{r._time}, {r._temp}".format(r=self)
        if format_spec == 'data':
            # Return a nicely formatted content string for str()
            data_str = time.strftime('%d.%m.%Y', time.gmtime(self._time))
            time_str = time.strftime('%H:%M:%S', time.gmtime(self._time))
            return "{}°C on {} at {}".format(self._temp, data_str, time_str)
        return super(ExampleTemperatureRecord, self).__format__(format_spec)
```

(continues on next page)

(continued from previous page)

```
    def _encode_payload(self):
        return self._encode_struct('>Lh', self._time, self._temp)

    @classmethod
    def _decode_payload(cls, octets, errors):
        timestamp, temperature = cls._decode_struct('>Lh', octets)
        return cls(timestamp, temperature)

    # Make sure that _decode_payload gets only called with 6 octets
    _decode_min_payload_length = 6
    _decode_max_payload_length = 6

ndef.Record.register_type(ExampleTemperatureRecord)

record = ExampleTemperatureRecord(1468410873, 25)
octets = b''.join(ndef.message_encoder([record]))
print("encoded: {}".format(octets))

message = list(ndef.message_decoder(octets))
print("decoded: {}".format(message[0]))
```

```
encoded: b'\xd4\x10\x06nfcpy.org:x-tempW\x86+\xf9\x00\x19'
decoded: NDEF Example Temperature Record ID '' 25°C on 13.07.2016 at 11:54:33
```

## 3.3 Type Length Value Record

This record class demonstrates how `_decode_struct` and `_encode_struct` can be used for typical Type-Length-Value constructs. The notion 'BB+' is a slight extension of the `struct` module's format string syntax and means to decode or encode a 1 byte Type field, a 1 byte Length field and Length number of octets as Value. The `_decode_struct` method then returns just the Type and Value. The `_encode_struct` needs only the Type and Value arguments and takes the Length from Value. Another format string syntax extension, but not not used in the example, is a trailing '*' character. That just means that all remaining octets are returned as `bytes`.

This example also demonstrates how decode and encode error exceptions are generated with the `_decode_error` and `_encode_error` methods. These methods return an instance of `ndef.DecodeError` and `ndef.EncodeError` with the fully qualified class name followed by the expanded format string. Two similar methods, `_type_error` and `_value_error` may be used whenever a `TypeError` or `ValueError` shall be reported with the full classname in its error string. They do also check if the first word in the format string matches a data attribute name, and if, the string is joined with a '.' to the classname.

The `_decode_payload` method also shows the use of the errors argument. With 'strict' interpretation of errors the payload is expected to have the Type 1 TLV encoded in first place (although not a recommended design for TLV loops). The errors argument may also say 'relax' and then the order won't matter.

```
import ndef

class ExampleTypeLengthValueRecord(ndef.record.GlobalRecord):
    """An NDEF Record with carries a temperature and a timestamp."""

    _type = 'urn:nfc:ext:nfcpy.org:x-tlvs'

    def __init__(self, *args):
        # We expect each argument to be a tuple of (Type, Value) where Type
```

(continues on next page)

(continued from previous page)

```
        # is int and Value is bytes. So *args* will be a tuple of tuples.
        self._tlvs = args

    def _encode_payload(self):
        if sum([t for t, v in self._tlvs if t == 1]) != 1:
            raise self._encode_error("exactly one Type 1 TLV is required")
        tlv_octets = []
        for t, v in self._tlvs:
            tlv_octets.append(self._encode_struct('>BB+', t, v))
        return b''.join(tlv_octets)

    @classmethod
    def _decode_payload(cls, octets, errors):
        tlvs = []
        offset = 0
        while offset < len(octets):
            t, v = cls._decode_struct('>BB+', octets, offset)
            offset = offset + 2 + len(v)
            tlvs.append((t, v))
        if sum([t for t, v in tlvs if t == 1]) != 1:
            raise cls._encode_error("missing the mandatory Type 1 TLV")
        if errors == 'strict' and len(tlvs) > 0 and tlvs[0][0] != 1:
            errstr = 'first TLV must be Type 1, not Type {}'
            raise cls._encode_error(errstr, tlvs[0][0])
        return cls(*tlvs)

    # We need at least the 2 octets Type, Length for the first TLV.
    _decode_min_payload_length = 2

ndef.Record.register_type(ExampleTypeLengthValueRecord)

record = ExampleTypeLengthValueRecord((1, b'abc'), (5, b'xyz'))
octets = b''.join(ndef.message_encoder([record]))
print("encoded: {}".format(octets))

message = list(ndef.message_decoder(octets))
print("decoded: {}".format(message[0]))
```

```
encoded: b'\xd4\x10\nnfcpy.org:x-tlvs\x01\x03abc\x05\x03xyz'
decoded: NDEF Example Type Length Value Record ID '' PAYLOAD 10 byte
↪'0103616263050378797a'
```

CHAPTER 4

# Contributing

Thank you for considering contributing to **ndeflib**. There are many ways to help and any help is welcome.

## 4.1 Reporting issues

- Under which versions of Python does this happen? This is especially important if your issue is encoding related.
- Under which version of **ndeflib** does this happen? Check if this issue is fixed in the repository.

## 4.2 Submitting patches

- Include tests if your patch is supposed to solve a bug, and explain clearly under which circumstances the bug happens. Make sure the test fails without your patch.
- Include or update tests and documentation if your patch is supposed to add a new feature. Note that documentation is in two places, the code itself for rendering help pages and in the docs folder for the online documentation.
- Follow PEP 8 and PEP 257.

## 4.3 Development tips

- Fork the repository and clone it locally:

```
git clone git@github.com:your-username/ndeflib.git
cd ndeflib
```

- Create virtual environments for Python 2 an Python 3, setup the ndeftool package in develop mode, and install required development packages:

```
virtualenv python-2
python3 -m venv python-3
source python-2/bin/activate
python setup.py develop
pip install -r requirements-dev.txt
source python-3/bin/activate
python setup.py develop
pip install -r requirements-dev.txt
```

- Verify that all tests pass and the documentation is build:

```
tox
```

- Preferably develop in the Python 3 virtual environment. Running `tox` ensures tests are run with both the Python 2 and Python 3 interpreter but it takes some time to complete. Alternatively switch back and forth between versions and just run the tests:

```
source python-2/bin/activate
py.test
source python-3/bin/activate
py.test
```

- Test coverage should be close to 100 percent. A great help is the HTML output produced by coverage.py:

```
py.test --cov ndef --cov-report html
firefox htmlcov/index.html
```

- The documentation can be created and viewed loacally:

```
(cd docs && make html)
firefox docs/_build/html/index.html
```

CHAPTER 5

# License

The **ndeflib** is licensed under the Internet Systems Consortium ISC license. This is a permissive free software license functionally equivalent to the simplified BSD and the MIT license.

## 5.1 License text

# Index

## E



## F



## G



## H



## I



## L



## M



## N

## O



## P



## R



## S

## T



## U



## V



## W