# ndeflib documentation

***Release 0.1.1***

**Stephen Tiedemann**

January 04, 2017

**1 NDEF Decoding and Encoding** **3**
1.1 Message Decoder . . . 3
1.2 Message Encoder . . . 4
1.3 Record Class . . . 4

**2 Known Record Types** **7**
2.1 Text Record . . . 7
2.2 URI Record . . . 8
2.3 Smartposter Record . . . 8
2.4 Device Information Record . . . 10
2.5 Connection Handover . . . 11

**3 Adding Private Records** **17**
3.1 Record with no Payload . . . 17
3.2 Example Temperature Record . . . 18
3.3 Type Length Value Record . . . 19

**Python Module Index** **21**

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

`ndef.`**`message_decoder`**(*stream_or_bytes*, *errors='strict'*, *known_records=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_records* 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_records={}`.

**Parameters**

- **`stream_or_bytes`** (*`byte stream or bytes object`*) – message data octets
- **`errors`** (*`str`*) – error handling strategy, may be 'strict', 'relax' or 'ignore'
- **`known_records`** (*`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

ndef.**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** ndef.**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** ndef.**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** ndef.**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/%7Euser/'
>>> 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** ndef.**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** ndef.**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** ndef.**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** `ndef.`**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** `ndef.`**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** `ndef.`**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** ndef.**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.

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''

    @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)

    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
        # 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'
```

## n

ndef, 7

## A

action (ndef.SmartposterRecord attribute), 9
add_icon() (ndef.SmartposterRecord method), 9
add_undefined_data_element() (ndef.DeviceInformationRecord method), 10
alternative_carriers (ndef.HandoverInitiateRecord attribute), 15
alternative_carriers (ndef.HandoverMediationRecord attribute), 14
alternative_carriers (ndef.HandoverRequestRecord attribute), 12
alternative_carriers (ndef.HandoverSelectRecord attribute), 13

## C

carrier_data (ndef.HandoverCarrierRecord attribute), 15
carrier_type (ndef.HandoverCarrierRecord attribute), 15
collision_resolution_number (ndef.HandoverRequestRecord attribute), 12

## D

data (ndef.DeviceInformationRecord attribute), 10
data (ndef.HandoverCarrierRecord attribute), 15
data (ndef.HandoverInitiateRecord attribute), 14
data (ndef.HandoverMediationRecord attribute), 14
data (ndef.HandoverRequestRecord attribute), 12
data (ndef.HandoverSelectRecord attribute), 13
data (ndef.Record attribute), 5
data (ndef.SmartposterRecord attribute), 9
data (ndef.TextRecord attribute), 7
data (ndef.UriRecord attribute), 8
DeviceInformationRecord (class in ndef), 10

## E

encoding (ndef.TextRecord attribute), 7
error (ndef.HandoverSelectRecord attribute), 13

## H

HandoverCarrierRecord (class in ndef), 15
HandoverInitiateRecord (class in ndef), 14
HandoverMediationRecord (class in ndef), 13
HandoverRequestRecord (class in ndef), 11
HandoverSelectRecord (class in ndef), 12
hexversion (ndef.HandoverInitiateRecord attribute), 14
hexversion (ndef.HandoverMediationRecord attribute), 14
hexversion (ndef.HandoverRequestRecord attribute), 12
hexversion (ndef.HandoverSelectRecord attribute), 13

## I

icon (ndef.SmartposterRecord attribute), 9
icons (ndef.SmartposterRecord attribute), 9
iri (ndef.UriRecord attribute), 8

## L

language (ndef.TextRecord attribute), 7

## M

MAX_PAYLOAD_SIZE (ndef.Record attribute), 5
message_decoder() (in module ndef), 3
message_encoder() (in module ndef), 4
model_name (ndef.DeviceInformationRecord attribute), 10

## N

name (ndef.DeviceInformationRecord attribute), 10
name (ndef.HandoverCarrierRecord attribute), 15
name (ndef.HandoverInitiateRecord attribute), 14
name (ndef.HandoverMediationRecord attribute), 14
name (ndef.HandoverRequestRecord attribute), 12
name (ndef.HandoverSelectRecord attribute), 13
name (ndef.Record attribute), 5
name (ndef.SmartposterRecord attribute), 9
name (ndef.TextRecord attribute), 7
name (ndef.UriRecord attribute), 8
ndef (module), 1, 6–8, 10, 11, 16

## R

Record (class in ndef), 4

register_type() (ndef.Record class method), 6
resource (ndef.SmartposterRecord attribute), 9
resource_size (ndef.SmartposterRecord attribute), 9
resource_type (ndef.SmartposterRecord attribute), 9
RFC
    RFC 2046, 5
    RFC 2141, 5
    RFC 3986, 5
    RFC 3987, 8

## S

set_title() (ndef.SmartposterRecord method), 9
SmartposterRecord (class in ndef), 8

## T

text (ndef.TextRecord attribute), 7
TextRecord (class in ndef), 7
title (ndef.SmartposterRecord attribute), 9
titles (ndef.SmartposterRecord attribute), 9
type (ndef.DeviceInformationRecord attribute), 10
type (ndef.HandoverCarrierRecord attribute), 15
type (ndef.HandoverInitiateRecord attribute), 14
type (ndef.HandoverMediationRecord attribute), 14
type (ndef.HandoverRequestRecord attribute), 11
type (ndef.HandoverSelectRecord attribute), 12
type (ndef.Record attribute), 5
type (ndef.SmartposterRecord attribute), 9
type (ndef.TextRecord attribute), 7
type (ndef.UriRecord attribute), 8

## U

undefined_data_elements (ndef.DeviceInformationRecord attribute), 10
unique_name (ndef.DeviceInformationRecord attribute), 10
uri (ndef.UriRecord attribute), 8
UriRecord (class in ndef), 8
uuid_string (ndef.DeviceInformationRecord attribute), 10

## V

vendor_name (ndef.DeviceInformationRecord attribute), 10
version_info (ndef.HandoverInitiateRecord attribute), 15
version_info (ndef.HandoverMediationRecord attribute), 14
version_info (ndef.HandoverRequestRecord attribute), 12
version_info (ndef.HandoverSelectRecord attribute), 13
version_string (ndef.DeviceInformationRecord attribute), 10
version_string (ndef.HandoverInitiateRecord attribute), 15
version_string (ndef.HandoverMediationRecord attribute), 14
version_string (ndef.HandoverRequestRecord attribute), 12
version_string (ndef.HandoverSelectRecord attribute), 13