

Web NFC

W3C Draft Community Group Report, 25 February, 2020

Kenneth Rohde Christiansen, Intel; Zoltan Kis, Intel; and François Beaufort, Google LLC (Editors)



Note: this EPUB edition does not represent the authoritative text of the specification; please consult the [original document](#) on the W3C Web Site.

Copyright © of the original documents: 2020 W3C® (MIT, ERCIM, Keio, Beihang).
All right reserved. W3C [liability](#), [trademark](#), and [document use](#) rules apply.

Latest editor's draft:

<https://w3c.github.io/web-nfc/>

Test suite:

<https://wpt.fyi/web-nfc/>

Editors:

Kenneth Rohde Christiansen ([Intel](#))

Zoltan Kis ([Intel](#))

François Beaufort ([Google LLC](#))

Former editor:

Alexander Shalamov ([Intel](#))

Participate:

[GitHub w3c/web-nfc](#)

[File a bug](#)

[Commit history](#)

[Pull requests](#)

[Copyright](#) © 2020 the Contributors to the Web NFC Specification, published by the [Web NFC Community Group](#) under the [W3C Community Contributor License Agreement \(CLA\)](#). A human-readable [summary](#) is available.

Near Field Communication (NFC) enables wireless communication between two devices at close proximity, usually less than a few centimeters. NFC is an international standard (ISO/IEC 18092) defining an interface and protocol for simple wireless interconnection of closely coupled devices operating at 13.56 MHz.

The hardware standard is defined in [*NFC Forum Technical Specifications*](#).

This document defines an API to enable selected use cases based on NFC technology. The current scope of this specification is [NDEF](#).

Low-level I/O operations (e.g. ISO-DEP, NFC-A/B, NFC-F) and Host-based Card Emulation (HCE) are **not** supported within the current scope.



Status of This Document

This specification was published by the [Web NFC Community Group](#). It is not a W3C Standard nor is it on the W3C Standards Track. Please note that under the [W3C Community Contributor License Agreement \(CLA\)](#) there is a limited opt-out and other conditions apply. Learn more about [W3C Community and Business Groups](#).

Implementers need to be aware that this specification is considered unstable. Implementers who are not taking part in the discussions will find the specification changing out from under them in incompatible ways. Vendors interested in implementing this specification before it eventually reaches the Candidate Recommendation phase should subscribe to the repository on GitHub and take part in the discussions.

1. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, *SHOULD*, and *SHOULD NOT* in this document are to be interpreted as described in [BCP 14](#) [[RFC2119](#)] [[RFC8174](#)] when, and only when, they appear in all capitals, as shown here.

This document defines conformance criteria that apply to a single product: the **UA** (user agent) that implements the interfaces it contains.

2. Introduction

This section is non-normative.

Web NFC user scenario is as follows: Hold a device in close proximity to a passively powered NFC tag, such as a plastic card or sticker, in order to read and/or write data.

NFC works using magnetic induction, meaning that the reader (an active, powered device) will emit a small electric charge which then creates a magnetic field. This field powers the passive device which turns it into electrical impulses to communicate data. Thus, when the devices are within range, a read is always performed (see NFC Analog Specification and NFC Digital Protocol, NFC Forum, 2006). The peer-to-peer connection works in a similar way, as the device periodically switches into a so-called initiator mode in order to scan for targets, then later to fall back into target mode. If a target is found, the data is read the same way as for tags.

As NFC is based on existing RFID standards, many NFC chipsets support reading RFID tags, but some of these are only supported by single vendors and not part of the NFC standards. As such, this document specifies ways to interact with the NFC Data Exchange Format (NDEF).



3. Terminology and conventions

The Augmented Backus-Naur Form (ABNF) notation used is specified in [\[RFC5234\]](#).

NFC stands for Near Field Communications, a short-range wireless technology operating at 13.56 MHz which enables communication between devices at a distance less than 10 cm. The NFC communications protocols and data exchange formats, and are based on existing radio-frequency identification (RFID) standards, including ISO/IEC 14443 and FeliCa. The NFC standards include ISO/IEC 18092[5] and those defined by the NFC Forum. See [NFC Forum Technical Specifications](#) for a complete listing.

An **NFC adapter** is the software entity in the underlying platform which provides access to NFC functionality implemented in a given hardware element (NFC chip). A device may have multiple NFC adapters, for instance a built-in one, and one or more attached via USB.

An **NFC tag** is a passive NFC device. The [NFC tag](#) is powered by magnetic induction when an active NFC device is in proximity range. An [NFC tag](#) that supports [NDEF](#) contains a single [NDEF message](#).

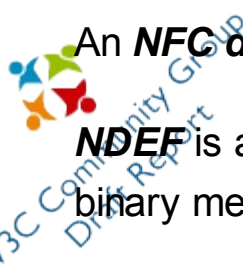
NOTE

The way of reading the message may happen through proprietary technologies, which require the reader and the tag to be of the same manufacturer. They may also expose an [NDEF](#) message.

An **NFC peer** is an active, powered device which can interact with other devices in order to exchange data using NFC.

[ISSUE 529](#): P2P support

As currently spec'ed, peer-to-peer is not supported.



An **NFC device** is either an [NFC peer](#), or an [NFC tag](#).

NDEF is an abbreviation for NFC Forum Data Exchange Format, a lightweight binary message format that is standardized in [\[NFC-NDEF\]](#).

An **NDEF message** encapsulates one or more application-defined [NDEF records](#). NDEF messages can be stored on an [NFC tag](#) or exchanged between NFC-enabled devices.

The term **NFC content** denotes all bytes sent to or received from an [NFC tag](#). In the current API it is synonym to [NDEF message](#).

4. The NFC Standard

This section is non-normative.

NFC is standardized in the NFC Forum and described in [[NFC-STANDARDS](#)].

4.1 NDEF compatible tag types

This section is non-normative.

The NFC Forum has mandated the support of five different tag types to be operable with NFC devices. The same is required on operating systems, such as Android.

In addition to that, the [MIFARE Standard](#) specifies a way for NDEF to work on top of the older [MIFARE Standard](#), which may be optionally supported by implementers.

A note about the NDEF mapping can be found here: [MIFARE Classic as NFC Type MIFARE Classic Tag](#).

1. **NFC Forum Type 1:** This tag is based on the ISO/IEC 14443-3A (NFC-A). The tags are rewritable and can be configured to become read-only. Memory size can be between 96 bytes and 2 Kbytes. Communication speed is 106 kbit/s. In contrast to all other types, these tags have no anti-collision protection for dealing with multiple tags within the NFC field.
2. **NFC Forum Type 2:** This tag is based on the ISO/IEC 14443-3A (NFC-A). The tags are rewritable and can be configured to become read-only. Memory size can be between 48 bytes and 2 Kbytes. Communication speed is 106 kbit/s.
3. **NFC Forum Type 3:** This tag is based on the Japanese Industrial Standard (JIS) X 6319-4 (ISO/IEC 18092), commonly known as FeliCa. The tags are preconfigured to be either rewritable or read-only. Memory is 2 kbytes. Communication speed is 212 kbit/s or 424 kbit/s.

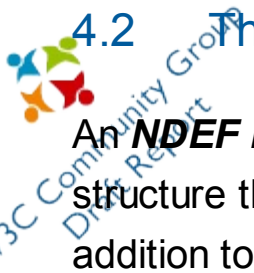


4. **NFC Forum Type 4:** This tag is based on the ISO/IEC 14443-4 A/B (NFC A, NFC B) and thus supports either NFC-A or NFC-B for communication. On top of that the tag may optionally support ISO-DEP (Data Exchange Protocol defined in ISO/IEC 14443 (ISO/IEC 14443-4:2008 Part 4: Transmission protocol). The tags are preconfigured to be either rewritable or read-only. Variable memory, up to 32 kbytes. Supports three different communication speeds 106 or 212 or 424 kbit/s.
5. **NFC Forum Type 5:** This tag is based on ISO/IEC 15693 (NFC-V) and allows reading and writing an NDEF message on an ISO/IEC 15693 RF tag that is accessible by long range RFID readers as well. The NFC communication is limited to short distance and may use the *Active Communication Mode* of ISO/IEC 18092 where the sending peer generates the field which balances power consumption and improves link stability. Variable memory, up to 64 kbytes. Communication speed is 26.48 kbit/s.
6. **MIFARE Standard:** This tag, often sold under the brand names MIFARE Classic or MIFARE Mini, is based on the ISO/IEC 14443-3A (also known as NFC-A, as defined in ISO/IEC 14443-3:2011, Part 3: Initialization and anticollision). The tags are rewritable and can be configured to become read-only. Memory size can be between 320 and 4 kbytes. Communication speed is 106 kbit/s.

NOTE

[MIFARE Standard](#) is not an NFC Forum type and can only be read by devices using NXP hardware. Support for reading and writing to tags based on the [MIFARE Standard](#) is thus non-nominative, but the type is included due to the popularity and use in legacy systems.

In addition to data types standardized for [NDEF records](#) by the NFC Forum, many commercial products such as bus cards, door openers may be based on the [MIFARE Standard](#) which requires specific NFC chips (same vendor of card and reader) in order to function.



4.2 The NDEF record and fields

An **NDEF record** is a part of an [NDEF message](#). Each record is a binary structure that contains a data payload, as well as associated type information. In addition to this, it includes information about how the data is structured, like payload size, whether the data is chunked over multiple records etc.

A generic record looks like the following:

Only the first three bytes (lines in figure) are mandatory. First the header byte, followed by the [TYPE LENGTH field](#) and [PAYLOAD LENGTH field](#), both of which may be zero.

The **TNF field** (bit 0-2, type name format) indicates the format of the type name and is often exposed by native NFC software stacks. The field can take binary values denoting the following NDEF record payload types:

TNF value	Description
0	Empty record
1	NFC Forum well-known type record
2	MIME type record
3	Absolute-URL record
4	NFC Forum external type record
5	Unknown record
6	Unchanged record
7	Reserved for future use

The **IL field** (bit 3, id length) indicates whether an [ID LENGTH field](#) is present. If the [IL field](#) is 0, then the [ID field](#) is not present either.

The **SR field** (bit 4, short record) indicates a short record, one with a payload length ≤ 255 bytes. Normal records can have payload lengths exceeding 255 bytes up to a maximum of 4 GB. Short records only use one byte to indicate length, whereas normal records use 4 bytes ($2^{32}-1$ bytes).

The **CF field** (bit 5, chunk flag) indicates whether the payload is [chunked](#) across multiple records.



NOTE

Web NFC turns all received chunked records into logical records and transparently chunks sent payload when that is needed.

The **ME field** (bit 6, message end) indicates whether this record is the last in the NDEF message.

The **MB field** (bit 7, message begin) indicates whether this record is the first of the NDEF message.

The **TYPE LENGTH field** is an unsigned 8-bit integer that denotes the byte size of the TYPE field.

The **TYPE field** is a globally unique and maintained identifier that describes the type of the PAYLOAD field in a structure, encoding and format dictated by value of the TNF field.


NOTE

The NFC Record Type Definition (RTD) Technical Specification requires that the TYPE field names *MUST* be compared in case-insensitive manner.

The **ID LENGTH field** is an unsigned 8-bit integer that denotes the byte size of the ID field.

The **ID field** is an identifier in the form of a URI reference ([RFC3986]) that is unique, and can be absolute or relative (in the latter case the application must provide a base URI). Middle and terminating chunk records *MUST NOT* have an ID field, other records *MAY* have it.

The **PAYLOAD LENGTH field** denotes the byte size of the PAYLOAD field. If the SR field is 1, its size is one byte, otherwise 4 bytes, representing an 8-bit or 32-bit unsigned integer, respectively.

 The **PAYLOAD field** carries the application bytes. Any internal structure of the data is opaque to NDEF. Note that in certain cases discussed later, this field *MAY* contain an [NDEF message](#) as data.

4.3 NDEF Record types

4.3.1 Empty NDEF record (TNF 0)

An **empty record's** [TYPE LENGTH field](#), [ID LENGTH field](#) and [PAYLOAD LENGTH field](#) *MUST* be 0, thus the [TYPE field](#), [ID field](#) and [PAYLOAD field](#) *MUST NOT* be present.

4.3.2 Well-known type records (TNF 1)

The NFC Forum has standardized a small set of useful sub record types in [[NFC-RTD](#)] (Resource Type Definition specifications) called **well-known type records**, for instance text, URL, media and others. In addition, there are record types designed for more complex interactions, such as smart posters (containing optional embedded records for url, text, signature and actions), and handover records.

The type information stored in the [TYPE field](#) of [well-known type records](#) can be of two kinds: [local types](#) and [global types](#).

4.3.2.1 Well-known local types

NFC Forum **local type** that are defined by the NFC Forum or by an application, and always start with lowercase character or a number. Those are usually short strings that are unique only within the local context of the containing record. They are used when the meaning of the types doesn't matter outside of the local

context of the containing record and when storage usage is a hard constraint. See [Smart poster](#) for an example on how local types are used.

NOTE

A [local type](#) is thus defined in terms of a containing record type, and thus doesn't need any namespacing. For this reason the same local type name can be used within another record type with different meaning and different payload type.

4.3.3 Well-known global types

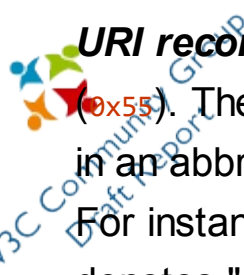
NFC Forum **global types** are defined and managed by the NFC Forum and usually start with an uppercase character. Examples: "T" for text, "U" for URL, "Sp" for smart poster, "Sig" for signature, "Hc" for handover carrier, "Hr" for handover request, "Hs" for handover select, etc.

4.3.3.1 Text record

The **Text record** is a [well-known type record](#) that is defined in the [\[NDEF-TEXT\]](#) specification. The [TNF field](#) is 1 and the [TYPE field](#) is "T" (0x54). The first byte of the [PAYLOAD field](#) is a status byte, followed by the [language tag](#) in US-ASCII encoding. The rest of the payload is the actual text, encoded either in UTF-8 or UTF-16, as indicated by the status byte as follows:

- Bits 0 to 5 define the length of the [language tag](#).
- Bit 6 is 0.
- If bit 7 is set, means the payload is encoded in UTF-8, otherwise in UTF-16.

4.3.3.2 URI record

 **URI record** is defined in [NDEF-URI]. The **TNF field** is **1** and the **TYPE field** is "u" (0x55). The first byte of the **PAYLOAD field** is a URI identifier code, in fact an index in an abbreviation table where the values are prepended to the rest of the URI. For instance the value **0** denotes no prepending, **1** denotes "http://www.", **0x04** denotes "https://" and so on. The rest of the payload contains the rest of the URI as a UTF-8 string (and if the first byte is **0**, then it denotes the whole URI).

The URI is defined in [RFC3987] and in fact is a UTF-8 encoded IRI that can be a URN or a URL.

4.3.3.3 Smart poster record

Smart poster is defined in [NDEF-SMARTPOSTER] to describe a given web content as an NDEF record that contains an **NDEF message** as payload, including the following records:

- A single mandatory **URI record** that refers to the **smart poster** content.
- Zero or more **Text records** that act as a **title record** related to the content. When there are more than one title record present, they *MUST* be with different **language tags**. Applications *SHOULD* select one **title record** for presentation to the end user.
- Zero or more **MIME type records** that act as **icon record** related to the content. The **MIME type** is usually "image/jpg", "image/png", "image/gif", or even "video/mpeg". Applications *SHOULD* select one **icon record** for presentation to the end user.
- One optional **type record** that has a **local type name** "t" specific to **smart poster** and the **PAYLOAD field** contains a UTF-8 encoded MIME type for the content referred to by the **URI record**.
- One optional **size record** that has **local type name** "s" specific to **smart poster** and the **PAYLOAD field** contains a 4-byte 32 bit unsigned integer that denotes the size of the object referred to by the URL in the **URI record** of the **smart poster**.



- One optional **action record** that has a local type name "act" specific to smart poster and the PAYLOAD field contains a single byte, whose value has the following meaning:

Value	Description
0	Do the action
1	Save for later
2	Open for editing
3..0xFF	Reserved for future use

There is no default action on the smart poster content if the action record is missing.


NOTE

At the time of NDEF standardization the value 0 ("do the action") was intended for use cases like send an SMS, make a call or launch browser. Similarly, the value 1, ("save the content for later processing") was intended for use cases like store the SMS in inbox, save the URL in bookmarks, or save the phone number to contacts. Also, the value 2 ("open for editing") was meant to open the smart poster content with a default application for editing.

Implementations don't need to implement any standardized behavior for the actions defined here. In this API it's up to the applications what actions they define (that may include the use cases above). However, Web NFC just provides the values.

The example below shows a smart poster record that embeds a text and a URL record.

4.3.3.4 Signature records

 **NDEF Signature** is defined [[NDEF-SIGNATURE](#)]. Its [TYPE field](#) contains "Sig" (0x53, 0x69, 0x67) and its [PAYLOAD field](#) contains version, signature and a certificate chain.

ISSUE 363: Support Signature records enhancement

As currently spec'ed, this is not supported.

4.3.3.5 Handover records

NFC handover is defined [[NFC-HANDOVER](#)] and the corresponding message structure that allows negotiation and activation of an alternative communication carrier, such as Bluetooth or WiFi. The negotiated communication carrier would then be used (separately) to perform certain activities between the two devices, such as sending photos to the other device, printing to a Bluetooth printer or streaming video to a television set.

ISSUE 364: Support Handover records enhancement

As currently spec'ed, this is not supported.

4.3.4 MIME type records (TNF 2)

The **MIME type records** are records that store binary data with associated [MIME type](#).

4.3.5 Absolute-URL records (TNF 3)

In **absolute-URL records** the [TYPE field](#) contains the [absolute-URL string](#), and not the payload.



NOTE

NOTE: Some platforms, like Windows Phone have stored additional data in the payload, but any payload data in these records are ignored by other platforms such as Android. On Android, reading such a record, will attempt to load the URL in Chrome and it is as such not intended for client applications.

4.3.6 External type records (TNF 4)

The NFC Forum **external type records** are for application specified data types and are defined in [NFC Record Type Definition \(RTD\) Technical Specification](#).

The **external type** is a URN with the prefix "urn:nfc:ext:" followed by the name of the owner [domain](#), adding a U+003A (:), then a non-zero type name, for instance "urn:nfc:ext:w3.org:atype", stored as "w3.org:atype" in the [TYPE field](#).

4.3.7 Unknown type records (TNF 5)

The **unknown records** are records that store opaque data without associated [MIME type](#), meaning that the `application/octet-stream` default [MIME type](#) MAY be assumed. The [\[NFC-NDEF\]](#) specification recommends that [NDEF](#) parsers store or forward the payload without processing it.

4.3.8 Unchanged type records (TNF 6)

The **unchanged records** are record chunks of a chunked data set, and is used for any, but the first record. A **chunked** payload is spread across multiple [NDEF records](#) that undergo the following rules:

- The initial chunk record has the [CF field](#) set, its [TYPE field](#) set to the type of the whole chunked payload and its [ID field](#) MAY be set to an identifier used



for the whole chunked payload. Its [PAYLOAD LENGTH field](#) denotes the size of the payload chunk in this record only.

- The middle chunk records have the [CF field](#) set, have the same [ID field](#) as the first chunk, their [TYPE LENGTH field](#) and [IL field](#) *MUST* be 0 and their [TNF field](#) *MUST* be 6 (unchanged).
- The terminating chunk record has this flag cleared, and in rest undergo the same rules as the middle chunk records.
- A chunked payload *MUST* be contained in a single [NDEF message](#), therefore the initial and middle chunk records cannot have the [ME field](#) set.

First record:

Intermediate record:

Last record:

Any implementation of Web NFC *MUST* transparently expose chunked records as single logical records.

5. Use Cases

This section is non-normative.

A few NFC user scenarios have been enumerated [here](#) and in the [Web NFC Use Cases](#) document. The rudimentary Web NFC interactions are the following.

5.1 Reading an [NFC tag](#)

Reading an [NFC tag](#) containing an [NDEF message](#), while the [Document](#) of the [top-level browsing context](#) using Web NFC is [visible](#). For instance, a web page instructs the user to tap an NFC tag, and then receives information from the tag.

5.2 Writing to an [NFC tag](#)


The user opens a web page which can write to an [NFC tag](#). The write operations may be one of the following:

1. Writing to a non-formatted [NFC tag](#).
2. Writing to an empty, but formatted [NFC tag](#).
3. Writing to an [NFC tag](#) which already contains an [NDEF message](#).
4. Writing to other, writable [NFC tags](#) (i.e. overwriting a generic tag).

NOTE

Note that an NFC write operation to an [NFC tag](#) always involves also a read operation.

5.3 Support for multiple NFC adapters

 Users may attach one or more external [NFC adapters](#) to their devices, in addition to a built-in adapter. Users may use either [NFC adapter](#).

6. Features

This section is non-normative.

High level features for the Web NFC specification include the following:

1. Support devices with single or multiple [NFC adapters](#). If there are multiple adapters present when invoking an NFC function then the UA operates all [NFC adapters](#) in parallel.
2. Support communication with passive (smart cards, tags, etc.) NFC devices.
3. Allow users to act on (e.g. read, write or transceive) discovered passive NFC devices, as well as access the payload which were read in the process as [NDEF messages](#).
4. Allow users to write a payload via [NDEF records](#) to compatible devices, such as writable tags, when they come in range, as [NDEF messages](#).

7. Examples

This section is non-normative.

This section shows how developers can make use of the various features of this specification.

7.1 Feature support

Detecting if Web NFC is supported can be done by checking NDEFReader and/or NDEFWriter objects. Note that this does not guarantee that NFC hardware is available.

EXAMPLE 1

```
if ('NDEFReader' in window) { /* ... Scan NDEF Tags */ }  
if ('NDEFWriter' in window) { /* ... Write NDEF Tags */ }
```

7.2 Write a text string

Writing a text string to an NFC tag is straightforward.

EXAMPLE 2

```
const writer = new NDEFWriter();  
writer.write(  
  "Hello World"  
)  
.then(() => {  
  console.log("Message written.");  
}).catch(error => {  
  console.log(`Write failed :-( try again: ${error}.`);  
});
```

7.3 Write a URL



In order to write an NDEF record of URL type, simply use NDEFMessage.

EXAMPLE 3

```
const writer = new NDEFWriter();
writer.write({
  records: [{ recordType: "url", data: "https://w3c.github.io/web-nfc/" }]
}).then(() => {
  console.log("Message written.");
}).catch(_ => {
  console.log("Write failed :-( try again.");
});
```

7.4 Handle scanning errors

This example shows what happens when scan promise rejects and onerror is fired.

EXAMPLE 4

```
const reader = new NDEFReader();
reader.scan().then(() => {
  console.log("Scan started successfully.");
  reader.onerror = event => {
    console.log("Error! Cannot read data from the NFC tag. Try a different one?");
  };
  reader.onreading = event => {
    console.log("NDEF message read.");
  };
}).catch(error => {
  console.log(`Error! Scan failed to start: ${error}.`);
});
```

7.5 Read data from tag, and write to empty ones

This example shows reading various different kinds of data which can be stored on a tag. If the tag is unformatted or contains an empty record, a text message is written with the value "Hello World".



EXAMPLE 5

```
const reader = new NDEFReader();
await reader.scan();
reader.onreading = event => {
  const message = event.message;

  if (message.records.length == 0 ||      // unformatted tag
      message.records[0].recordType == 'empty' ) { // empty record
    const writer = new NDEFWriter();
    writer.write({
      records: [{ recordType: "text", data: 'Hello World' }]
    });
    return;
  }

  const decoder = new TextDecoder();
  for (const record of message.records) {
    switch (record.recordType) {
      case "text":
        const textDecoder = new TextDecoder(record.encoding);
        console.log(`Text: ${textDecoder.decode(record.data)} (${record.lang})`);
        break;
      case "url":
        console.log(`URL: ${decoder.decode(record.data)}`);
        break;
      case "mime":
        if (record.mediaType === "application/json") {
          console.log(`JSON: ${JSON.parse(decoder.decode(record.data))}`);
        }
        else if (record.mediaType.startsWith('image/')) {
          const blob = new Blob([record.data], { type: record.mediaType });

          const img = document.createElement("img");
          img.src = URL.createObjectURL(blob);
          img.onload = () => window.URL.revokeObjectURL(this.src);

          document.body.appendChild(img);
        }
        else {
          console.log(`Media not handled`);
        }
        break;
      default:
        console.log(`Record not handled`);
    }
  }
}
```



7.6 Save and restore game progress with an NFC tag

Filtering of relevant data sources can be done by the use of the [NDEFScanOptions](#). Below we use the custom record identifier "my-game-progress" as a relative URL so that when we read the data, we immediately update the game progress by issuing a write with a custom NDEF data layout.

EXAMPLE 6

```
const reader = new NDEFReader();  
await reader.scan({ id: "my-game-progress" });  
reader.onreading = async event => {  
  console.log(`Game state: ${JSON.stringify(event.message.records)}`);  
  
  const encoder = new TextEncoder();  
  const newMessage = {  
    records: [{  
      id: "my-game-progress",  
      recordType: "mime",  
      mediaType: "application/json",  
      data: encoder.encode(JSON.stringify({  
        level: 3,  
        points: 4500,  
        lives: 3  
      })))  
    }]  
  };  
  const writer = new NDEFWriter();  
  await writer.write(newMessage);  
  console.log("Message written");  
};
```

7.7 Write and read JSON (serialized and deserialized)

EXAMPLE 7

```
const reader = new NDEFReader();
await reader.scan({
  mediaType: "application/*json"
});
reader.onreading = event => {
  const decoder = new TextDecoder();
  for (const record of event.message.records) {
    if (record.mediaType === 'application/json') {
      const json = JSON.parse(decoder.decode(record.data));
      const article = /^[aeio]/i.test(json.title) ? "an" : "a";
      console.log(`${json.name} is ${article} ${json.title}`);
    }
  }
};

const writer = new NDEFWriter();
const encoder = new TextEncoder();
writer.write({
  records: [
    {
      recordType: "mime",
      mediaType: "application/json",
      data: encoder.encode(JSON.stringify({
        name: "Benny Jensen",
        title: "Banker"
      })))
    },
    {
      recordType: "mime",
      mediaType: "application/json",
      data: encoder.encode(JSON.stringify({
        name: "Zoey Braun",
        title: "Engineer"
      })))
    }
  ]
});
```

Writing data requires tapping an [NFC tag](#). If data should be read during the same tap, we need to set the [ignoreRead](#) property to `false`.

EXAMPLE 8

```
const reader = new NDEFReader();
reader.scan().then(() => {

  reader.onreading = event => {
    const decoder = new TextDecoder();
    for (const record of event.message.records) {
      console.log("Record type: " + record.recordType);
      console.log("MIME type:   " + record.mediaType);
      console.log("=== data ===\n" + decoder.decode(record.data));
    }
  };

  const writer = new NDEFWriter();
  return writer.write("Writing data is fun!", { ignoreRead: false });

}).catch(error => {
  console.log(`Write failed :-( try again: ${error}.`);
});
```

7.9 Stop listening to NDEF messages

Read NDEF messages for 3 seconds by using [signal](#).



EXAMPLE 9

```
const reader = new NDEFReader();
const controller = new AbortController();

await reader.scan({ signal: controller.signal });
reader.onreading = event => {
  console.log("NDEF message read.");
};

controller.signal.onabort = event => {
  console.log("We're done waiting for NDEF messages.");
};

// Stop listening to NDEF messages after 3s.
setTimeout(() => controller.abort(), 3000);
```

7.10 Write a smart poster message



EXAMPLE 10

```
const writer = new NDEFWriter();
const encoder = new TextEncoder();
writer.write({ records: [
  {
    recordType: "smart-poster", // Sp
    data: { records: [
      {
        recordType: "url", // URL record for the Sp content
        data: "https://my.org/content/19911"
      },
      {
        recordType: "text", // title record for the Sp content
        data: "Funny dance"
      },
      {
        recordType: ":t", // type record, a local type to Sp
        data: encoder.encode("image/gif") // MIME type of the Sp content
      },
      {
        recordType: ":s", // size record, a local type to Sp
        data: new Uint32Array([4096]) // byte size of Sp content
      },
      {
        recordType: ":act", // action record, a local type to Sp
        // do the action, in this case open in the browser
        data: new Uint8Array([0])
      },
      {
        recordType: "mime", // icon record, a MIME type record
        mediaType: "image/png",
        data: await (await fetch("icon1.png")).arrayBuffer()
      },
      {
        recordType: "mime", // another icon record
        mediaType: "image/jpg",
        data: await (await fetch("icon2.jpg")).arrayBuffer()
      }
    ]}
  }
]});
```



7.11 Read an external record with an NDEF message as payload

External type records can be used to create application defined records. These records may contain an [NDEF message](#) as payload, with its own [NDEF records](#), including [local types](#) that are used in the context of the application.

Note that the [smart poster](#) record type also contains an [NDEF message](#) as payload.

As NDEF gives no guarantee on the ordering of records, using an external type record with an [NDEF message](#) as payload, can be useful for encapsulating related data.

This example shows how to read an external record for social posts, which contains an [NDEF message](#), containing a text record and a record with the [local type](#) "act" (action), with definition borrowed from [smart poster](#), but used in local application context.



EXAMPLE 11

```
const reader = new NDEFReader();
await reader.scan({ recordType: "example.com:smart-poster" });
reader.onreading = event => {
  const externalRecord = event.message.records.find(
    record => record.type == "example.com:smart-poster"
  );

  let action, text;

  for (const record of externalRecord.toRecords()) {
    if (record.recordType == "text") {
      const decoder = new TextDecoder(record.encoding);
      text = decoder.decode(record.data);
    } else if (record.recordType == ":act") {
      action = record.data.getUint8(0);
    }
  }

  switch (action) {
    case 0: // do the action
      console.log(`Post "${text}" to timeline`);
      break;
    case 1: // save for later
      console.log(`Save "${text}" as a draft`);
      break;
    case 2: // open for editing
      console.log(`Show editable post with "${text}"`);
      break;
  }
};
```

7.12 Write an external record with an NDEF message as payload

External type records can be used to create application defined records that may even contain an [NDEF message](#) as payload.



EXAMPLE 12

```
const writer = new NDEFWriter();
writer.write({ records: [
  {
    recordType: "example.game:a",
    data: {
      records: [
        {
          recordType: "url",
          data: "https://example.game/42"
        },
        {
          recordType: "text",
          data: "Game context given here"
        },
        {
          recordType: "mime",
          mediaType: "image/png",
          data: getImageBytes(fromURL)
        }
      ]
    }
  }
]
});
```

7.13 Write and read unknown records inside an external record

Unknown type records may be useful inside external type records as developers know what they represent and therefore can avoid specifying the mime type.



EXAMPLE 13

```
const encoder = new TextEncoder();
const writer = new NDEFWriter();
writer.write({ records: [
  {
    recordType: "example.com:shoppingItem", // External record
    data: {
      records: [
        {
          recordType: "unknown", // Shopping item name
          data: encoder.encode("Food")
        },
        {
          recordType: "unknown", // Shopping item description
          data: encoder.encode("Provide nutritional support for an organism.")
        }
      ]
    }
  }
}]);
```

EXAMPLE 14

```
const reader = new NDEFReader();
await reader.scan({ recordType: "example.com:shoppingItem" });
reader.onreading = event => {
  const shoppingItemRecord = event.message.records[0];
  if (!shoppingItemRecord) {
    return;
  }

  const [nameRecord, descriptionRecord] = shoppingItemRecord.toRecords();

  const decoder = new TextDecoder();
  console.log("Item name: " + decoder.decode(nameRecord.data));
  console.log("Item description: " + decoder.decode(descriptionRecord.data));
};
```

8. Data Representation

8.1 The *NDEFMessage* interface

The content of any NDEF message is exposed by the *NDEFMessage* interface:

WebIDL

```
[SecureContext, Exposed=Window]
interface NDEFMessage {
    constructor(NDEFMessageInit messageInit);
    readonly attribute FrozenArray<NDEFRecord> records;
};

dictionary NDEFMessageInit {
    required sequence<NDEFRecordInit> records;
};
```

The *records* property represents a list of NDEF records defining the NDEF message.

The *NDEFMessageInit* dictionary is used to initialize an NDEF message.

8.2 The *NDEFRecord* interface

The content of any NDEF record is exposed by the *NDEFRecord* interface:

```

typedef (DOMString or BufferSource or NDEFMessageInit) NDEFRecordDataSource;

[SecureContext, Exposed=Window]
interface NDEFRecord {
    constructor(NDEFRecordInit recordInit);

    readonly attribute USVString recordType;
    readonly attribute USVString? mediaType;
    readonly attribute USVString? id;
    readonly attribute DataView? data;

    readonly attribute USVString? encoding;
    readonly attribute USVString? lang;

    sequence<NDEFRecord>? toRecords();
};

dictionary NDEFRecordInit {
    required USVString recordType;
    USVString mediaType;
    USVString id;

    USVString encoding;
    USVString lang;

    NDEFRecordDataSource data;
};

```

The *mediaType* property represents the [MIME type](#) of the [NDEF record](#) payload.

The *recordType* property represents the [NDEF record](#) types.

The *id* property represents the **record identifier**, which is an absolute or relative URL. The required uniqueness of the identifier is guaranteed only by the generator, not by this specification.



NOTE

The NFC NDEF specifications uses the terms "message identifier" and "payload identifier" instead of [record identifier](#), but the identifier is tied to each record and not the message (collection of records), and it may be present when no payload is.

The *encoding* attribute represents the [encoding name](#) used for encoding the payload in the case it is textual data.

The *lang* attribute represents the [language tag](#) of the payload in the case that was encoded.

A **language tag** is a [string](#) that matches the production of a [Language-Tag](#) defined in the [\[BCP47\]](#) specifications (see the [IANA Language Subtag Registry](#) for an authoritative list of possible values). That is, a language range is composed of one or more **subtags** that are delimited by a U+002D HYPHEN-MINUS ("-"). For example, the 'en-AU' language range represents English as spoken in Australia, and 'fr-CA' represents French as spoken in Canada. Language tags that meet the validity criteria of [\[RFC5646\]](#) section 2.2.9 that can be verified without reference to the IANA Language Subtag Registry are considered structurally valid.

The *data* property represents the [PAYLOAD field](#) data.

The *toRecords()* method, when invoked, *MUST* return the result of running [convert NDEFRecord.data bytes](#) with the [NDEF Record](#).

The *NDEFRecordInit* dictionary is used to initialize an [NDEF record](#) with its [record type](#) *recordType*, and optional [record identifier](#) *id* and payload data *data*.

Additionally, there are additional optional fields that are only applicable for certain [record types](#):

- "[mime](#)": Optional [MIME type](#) *mediaType*.
- "[text](#)": Optional [encoding label](#) *encoding* and [language tag](#) *lang*.

The mapping from data types of an [NDEFRecordInit](#) to [NDEF record](#) types is presented in the algorithmic steps which handle the data and described in the [§ 9.12 Parsing content](#) and [§ 9.9 Writing content](#) sections.

To **convert *NDEFRecord.data bytes*** given a *record*, run these steps:

1. Let *bytes* be the value of record's [data](#) attribute.
2. Let *recordType* be the value of record's [recordType](#) attribute.
3. If the *recordType* value is "[smart-poster](#)", or if running [validate external type](#) on *recordType* returns [true](#), then return the result of running [parse records from bytes](#) on *bytes*.
4. Otherwise, [throw](#) a "[NotSupportedError](#)" [DOMException](#) and abort these steps.

8.3 The *record type* string

This string defines the allowed record types for an [NDEFRecord](#). The [§ 8.4 Data mapping](#) section describes how it is mapped to [NDEF record](#) types.

A standardized ***well known type name*** can be one of the following:

The "*empty*" string

The value representing an [empty](#) [NDEFRecord](#).

The "*text*" string

The value representing a [Text record](#).

The "*url*" string

The value representing a [URI record](#).

The "*smart-poster*" string

The value representing a [Smart poster](#) record.

The "*absolute-url*" string

The value representing an [absolute-URL record](#).

The "*mime*" string

The value representing a [MIME type record](#).

The "*unknown*" string

The value representing an [unknown record](#).

In addition to [well known type names](#) it is also possible for organizations to create a custom **external type name**, which is a string consisting of a [domain](#) name and a custom type name, separated by a colon U+003A (:).

Applications *MAY* also use a **local type name**, which is a string that *MUST* start with lowercase character or a number, representing a type for an NFC Forum [local type](#). It is typically used in a record of an [NDEFMessage](#) that is the payload of a parent [NDEFRecord](#), for instance in a [smart poster](#). The context of the [local type](#) is the parent record whose payload is the [NDEFMessage](#) to which this record belongs and the [local type name](#) *SHOULD NOT* conflict with any other type names used in that context.

Any implementation of Web NFC *MUST* transparently expose chunked records as single logical records, therefore [unchanged records](#) are not explicitly represented.

Two [well-known type records](#) (including any NFC Forum [local type](#) and any NFC Forum [global type](#)) *MUST* be compared character by character in case-sensitive manner.

Two external types *MUST* be compared character by character, in case-insensitive manner.

The binary representation of any [well-known type record](#) and [external type](#) *MUST* be written as a relative URI (RFC 3986), omitting the namespace identifier (NID) "nfc" and namespace specific string (NSS) "wkt" and "ext", respectively, i.e. omitting the "urn:nfc:wkt:" and "urn:nfc:ext:" prefixes. For instance, "urn:nfc:ext:company.com:a" is stored as "company.com:a" and the [well-known type records](#) of a [Text record](#) is "urn:nfc:wkt:T", but it is stored as "T".

8.4 Data mapping

The mapping from data types of an [NDEFRecordInit](#) to [NDEF record](#) types, as used in the § 9.9 [Writing content](#) section is as follows:



W3C Confidential

recordType	mediaType	data	record type	TNF field	TYPE field
"empty"	unused	unused	Empty record	0	unused
"text"	unused	BufferSource or DOMString	Well-known type record	1	"T"
"url"	unused	DOMString	Well-known type record	1	"U"
"smart-poster"	unused	NDEFMessageInit	Well-known type record	1	"Sp"
local type name prefixed by a colon U+003A (:), e.g., ":act", ":s", and ":t"	unused	BufferSource or NDEFMessageInit	Local type record*	1	local type name, e.g., "act", "s", and "t"
"mime"	MIME type	BufferSource	MIME type record	2	MIME type
"absolute-url"	unused	DOMString url	Absolute-URL record	3	Absolute-URL
external type name	unused	BufferSource or NDEFMessageInit	External type record	4	external type name
"unknown"	unused	BufferSource	Unknown record	5	unused

* A [local type](#) record has to be embedded with the [NDEFMessage](#) payload of another record.

The mapping from [NDEF record](#) types to [NDEFRecord](#), as used for incoming [NDEF messages](#) described in the [§ 9.12 Parsing content](#) section, is as follows.

record type	TNF field	TYPE field	recordType	mediaType
Empty record	0	unused	"empty"	null
Well-known type record	1	"T"	"text"	null



<u>Well-known type record</u>	1	"u"	"url"	null
<u>Well-known type record</u>	1	"Sp"	"smart-poster"	null
<u>Local type record</u> *	1	<u>local type name</u> , e.g., "act", "s", and "t"	<u>local type name</u> prefixed by a colon U+003A (:), e.g., ":act", ":s", and ":t"	null
<u>MIME type record</u>	2	<u>MIME type</u>	"mime"	The <u>MIME type</u> used in the NDEF record
<u>Absolute-URL record</u>	3	URL	"absolute-url"	null
<u>External type record</u>	4	<u>external type name</u>	<u>external type name</u>	null
<u>Unknown record</u>	5	<i>unused</i>	"unknown"	null

9. The NDEFReader and NDEFWriter objects

The objects provide a way for the [browsing context](#) to use NFC functionality. They allow for writing [NDEF messages](#) to [NFC tags](#) within range, and to act on incoming [NDEF messages](#) from [NFC tags](#).

WebIDL

```
typedef (DOMString or BufferSource or NDEFMessageInit) NDEFMessageSource;

[SecureContext, Exposed=Window]
interface NDEFWriter {
    constructor();

    Promise<void> write(NDEFMessageSource message, optional NDEFWriteOptions options={})
};

[SecureContext, Exposed=Window]
interface NDEFReader : EventTarget {
    constructor();

    attribute EventHandler onerror;
    attribute EventHandler onreading;

    Promise<void> scan(optional NDEFScanOptions options={});
};

[SecureContext, Exposed=Window]
interface NDEFReadingEvent : Event {
    constructor(DOMString type, NDEFReadingEventInit readingEventInitDict);

    readonly attribute DOMString serialNumber;
    [SameObject] readonly attribute NDEFMessage message;
};

dictionary NDEFReadingEventInit : EventInit {
    DOMString? serialNumber = "";
    required NDEFMessageInit message;
};
```

The [NDEFMessageSource](#) is a union type representing argument types accepted by the [write\(\)](#) method.

The **NDEFReadingEvent** is the event being dispatched on new NFC readings. The **serialNumber** property represents the serial number of the device used for anti-collision and identification, or empty string in case none is available. The **message** is an **NDEFMessage** object.

NDEFReadingEventInit is used to initialize a new event with a serial number and the **NDEFMessageInit** data via the **message** member. If **serialNumber** is not present or is **null**, empty string will be used to init the event.

NOTE

Though most tags will have a stable unique identifier (UID), not all have one and some tags even create a random number on each read. The serial number usually consists of 4 or 7 numbers, separated by **:**.

The **NDEFWriter** is an object used for writing data to an **NFC tag**.

An **NDEFWriter** object has the following internal slots:

Internal Slot	Initial value	Description (<i>non-normative</i>)
[[WriteOptions]]	null	The NDEFWriteOptions value for writer.
[[WriteMessage]]	null	The NDEFMessage to be written. It is initially unset.

The **NDEFReader** is an object used for reading data when a device, such as a tag, is within the magnetic induction field.

An **NDEFReader** object has the following internal slots:

Internal Slot	Initial value	Description (<i>non-normative</i>)
[[Id]]	undefined	The NDEFScanOptions.id value.
[[RecordType]]	undefined	The NDEFScanOptions.recordType value.
[[MediaType]]	undefined	The NDEFScanOptions.mediaType value.
[[Signal]]	undefined	The NDEFScanOptions.signal to abort the operation.



NOTE

Note that the internal slots of [NDEFReader](#) come from the *options* passed to [NDEFReader.scan\(\)](#). Therefore there is maximum one filter associated with any given [NDEFReader](#) object and successive invocations of [NDEFReader.scan\(\)](#) with new *options* will replace existing filters.

The *onreading* is an [EventHandler](#) which is called to notify that new reading is available.

The *onerror* is an [EventHandler](#) which is called to notify that an error happened during reading.

9.1 NFC state associated with the settings object

The [relevant settings object](#) of the [active document](#) of a [browsing context](#) which supports NFC has an associated **NFC state** record with the following [internal slots](#):

Internal Slot	Initial value	Description (<i>non-normative</i>)
[[Suspended]]	false	A boolean flag indicating whether NFC functionality is suspended or not, initially false.
[[ActivatedReaderList]]	empty set	A set of NDEFReader instances.
[[PendingWrite]]	empty	A <i><promise, writer></i> tuple where <i>promise</i> holds a pending Promise and <i>writer</i> holds an NDEFWriter .

The **activated reader objects** is the value of the [\[\[ActivatedReaderList\]\]](#) internal slot.

The **pending write tuple** is the value of the [\[\[PendingWrite\]\]](#) internal slot.

NFC is *suspended* if the [\[\[Suspended\]\]](#) internal slot is **true**.

To **suspend NFC**, set the [\[\[Suspended\]\]](#) internal slot to **true**.

To **resume NFC**, set the [\[\[Suspended\]\]](#) internal slot to **false**.

NOTE

Internal slots are used only as a notation in this specification, and implementations do not necessarily have to map them to explicit internal properties.

9.2 Handling NFC adapters

Implementations *MAY* use multiple [NFC adapters](#) according to the algorithmic steps described in this specification.

9.3 Obtaining permission

The [Web NFC permission name](#) is [defined](#) as **"nfc"**.

To **obtain permission**, run these steps:

1. Run the [query a permission](#) steps for the [Web NFC permission name](#) until completion.
 1. If it resolved with **"granted"** (i.e. permission has been granted to the [origin](#) and [global object](#) using the [Permissions](#) API), return **true**.
 2. Otherwise, if it resolved with **"prompt"**, then optionally [request permission](#) from the user for the [Web NFC permission name](#). If that is granted, return **true**.

ISSUE 482: permissions-related example in explainer encourages bad assumptions about browser behavior

Origin Trial

TAG feedback

The request permission steps are not yet clearly defined. At this point the UA asks the user about the policy to be used with the Web NFC permission name for the given origin and global object, if the user grants permission, return **true**.

2. Return **false**.

9.4 Handling visibility change

When the user agent determines that the **visibility state** of the responsible document of the current settings object changes, it must run these steps:

1. Let *document* be the responsible document of the current settings object.
2. If *document's* visibility state is "**visible**", resume NFC and abort these steps.
3. Otherwise, suspend NFC and attempt to abort a pending write operation.

The term **suspended** refers to NFC operations being suspended, which means that no NFC content is written by NDEFWriters, and no received NFC content is presented to any NDEFReader while being suspended.

9.5 Aborting pending write operation

To attempt to **abort a pending write operation** on an environment settings object, perform the following steps:

1. If there is no pending write tuple *tuple*, abort these steps.
2. If *tuple's* writer has already initiated an ongoing NFC data transfer, abort these steps.
3. Reject *tuple's* promise with an "AbortError" DOMException and abort these steps.



NOTE

Rejecting the promise will clear the [pending write tuple](#).

9.6 Releasing NFC

To **release NFC** on an [environment settings object](#), perform the following steps:

1. [Suspend NFC](#).
2. Attempt to [abort a pending write operation](#).
3. Stop the [dispatch NFC content](#) steps.
4. Clear the [activated reader objects](#).
5. Release the NFC resources associated with *nfc* on the underlying platform.

The UA must [release NFC](#) given the document's [relevant settings object](#) as additional [unloading document cleanup steps](#).

9.7 The *NDEFWriteOptions* dictionary

WebIDL

```
dictionary NDEFWriteOptions {  
  boolean ignoreRead = true;  
  boolean overwrite = true;  
  AbortSignal? signal;  
};
```

When the value of the *ignoreRead* property is *true*, the [NFC reading algorithm](#) will skip reading [NFC tags](#) for the [activated reader objects](#).

When the value of the *overwrite* property is *false*, the [write algorithm](#) will read the [NFC tag](#) regardless of the *ignoreRead* value to determine if it has [NDEF](#) records on it, and if yes, it will not execute any pending write.

The **signal** property allows to abort the [write\(\)](#) operation.

9.8 The **NDEFScanOptions** dictionary

To describe which messages an application is interested in, the [NDEFScanOptions](#) dictionary is used:

WebIDL

```
dictionary NDEFScanOptions {
  USVString id;
  USVString recordType;
  USVString mediaType;
  AbortSignal? signal;
};
```

The **signal** property allows to abort the [scan\(\)](#) operation.

The **id** property denotes the string value which is used for matching the [record identifier](#) of each [NDEFRecord](#) object in an [NDEF message](#). If the dictionary member is [not present](#), then it will be ignored by the [NFC listen algorithm](#).

The **recordType** property denotes the string value which is used for matching the [record type](#) of each [NDEFRecord](#) object in an [NDEF message](#). If the dictionary member is [not present](#), then it will be ignored by the [NFC listen algorithm](#).

The **mediaType** property denotes the [match pattern](#) which is used for matching the [mediaType](#) property of each [NDEFRecord](#) object in an [NDEF message](#).

EXAMPLE 15: Filter accepting only JSON content

```
const options = {
  mediaType: "application/*json" // any JSON-based MIME type
}
```




EXAMPLE 16: Filter which only accepts binary content for a custom record identifier

```
const options = {  
  id: "my-restaurant-daily-menu",  
  mediaType: "application/octet-stream"  
}
```

9.9 Writing content

This section describes how to write an [NDEF message](#) to an [NFC tag](#) when it is next time in proximity range before a timer expires. At any time there is a maximum of one [NDEF message](#) that can be set for writing for an [origin](#) until the current message is sent or the write is aborted.

9.9.1 The write() method

The `NDEFWriter.write` method, when invoked, *MUST* run the **write a message** algorithm:

1. Let *p* be a new [Promise](#) object.
2. Let *message* be the first argument.
3. Let *options* be the second argument.
4. Let *signal* be the *options*' dictionary member of the same name if present, or `null` otherwise.
5. If *signal*'s [aborted flag](#) is set, then reject *p* with an "[AbortError](#)" [DOMException](#) and return *p*.
6. If *signal* is not `null`, then [add the following abort steps](#) to *signal*:
 1. Run the [abort a pending write operation](#) on the [environment settings object](#).
7. [React](#) to *p*:



1. If p was settled (fulfilled or rejected), then clear the pending write tuple if it exists.

8. Return p and run the following steps in parallel:

1. If the obtain permission steps return `false`, then reject p with a `"NotAllowedError"` DOMException and abort these steps.
2. If there is no underlying NFC Adapter, or if a connection cannot be established, then reject p with a `"NotSupportedError"` DOMException and abort these steps.
3. If the UA is not allowed to access the underlying NFC Adapter (e.g. a user preference), then reject p with a `"NotReadableError"` DOMException and abort these steps.
4. If pushing data is not supported by the underlying NFC Adapter, then reject p with a `"NotSupportedError"` DOMException and abort these steps.
5. An implementation *MAY* reject p with a `"NotSupportedError"` DOMException and abort these steps.

NOTE

The UA might abort message write at this point. The reasons for termination are implementation details. For example, the implementation might be unable to support the requested operation.

6. Let *output* be the notation for the NDEF message to be created by UA, as the result of passing *message* to create NDEF message. If this throws an exception, reject p with that exception and abort these steps.
7. Attempt to abort a pending write operation.

NOTE

A write replaces all previously configured write operations.

8. Set `this.[WriteOptions]` to *options*.
9. Set `this.[WriteMessage]` to *output*.
10. Set pending write tuple to (`this`, *p*).
11. Run the start the NFC write steps whenever an NFC tag device comes within communication range.

NOTE

If NFC is suspended, continue waiting until promise is aborted by the user or an NFC tag comes within communication range.

To **start the NFC write**, run these steps:

1. Let *p* be the pending write tuple's promise.
2. Let *writer* be the pending write tuple's writer.
3. Let *options* be *writer*.`[WriteOptions]`.
4. If the NFC tag in proximity range does not expose NDEF technology for formatting or writing, then reject *p* with a "`NotSupportedError`" `DOMException` and return *p*.
5. Verify that NFC is not suspended.
6. In case of success, run the following steps:
 1. If *device* is an NFC tag and if *options*'s `overwrite` is `false`, read the tag to check whether there are NDEF records on the tag. If yes, then reject *p* with a "`NotAllowedError`" `DOMException` and return *p*.
 2. Let *output* be *writer*.`[WriteMessage]`.
 3. Initiate data transfer to *device* using *output* as buffer, using the NFC adapter in communication range with *device*.

NOTE

If the [NFC tag](#) in proximity range is unformatted and [NDEF](#)-formatable, format it and write *output* as buffer.

NOTE

Multiple adapters should be used sequentially by users. There is very little likelihood that a simultaneous tap will happen on two or multiple different and connected [NFC adapters](#). If it happens, the user will likely need to repeat the taps until success, preferably one device at a time. The error here gives an indication that the operation needs to be repeated. Otherwise the user may think the operation succeeded on all connected [NFC adapters](#).

4. If the transfer fails, reject *p* with "[NetworkError](#)" [DOMException](#) and abort these steps.
5. When the transfer has completed, resolve *p*.

9.10 Creating NDEF message

To **create NDEF message** given a *source* run these steps:

1. Switch on *source*'s type:

[DOMString](#)

- Let *textRecord* be an [NDEFRecord](#) initialized with its *recordType* set to "[text](#)" and *data* set to *source*.
- Let *records* be the list « *textRecord* ».
- Set *source*'s records to *records*.

[BufferSource](#)

- Let *mimeRecord* be an [NDEFRecord](#) initialized with its *recordType* set to "mime", *data* set to *source*, and *mediaType* set to "application/octet-stream".
- Let *records* be the list « *mimeRecord* ».
- Set *source*'s records to *records*.

NDEFMessageInit

- If *source*'s records [is empty](#), throw a [TypeError](#) and abort these steps.

unmatched type

- [throw](#) a [TypeError](#) and abort these steps.

2. Let *output* be the notation for the [NDEF message](#) to be created by the UA as a result of these steps.
3. [For each](#) *record* in the [list](#) *source*'s records, run the following steps:
 1. Let *ndef* be the result of running [create NDEF record](#) given *record*, or make sure the underlying platform provides equivalent values to *ndef*. If the algorithm throws an exception *e*, reject *promise* with *e* and abort these steps.
 2. Add *ndef* to *output*.
4. Return *output*.

9.10.1 Creating NDEF record

To **create NDEF record** given *record*, run these steps:

1. Let *ndef* be the representation of an [NDEF record](#) to be created by the UA.
2. If *record*'s [id](#) is not [undefined](#):
 - Let *identifier* be *record*'s [id](#).
 - Set *ndef*'s [IL field](#) to 1.
 - Set *ndef*'s [ID LENGTH field](#) to the length of *identifier*.
 - Set *ndef*'s [ID field](#) to *identifier*.

3. If *record*'s recordType starts with a colon U+003A (:):

- If *record* is not a payload to another NDEF record, reject *promise* with a TypeError and abort these steps.
- If running the validate local type steps on *record*'s recordType returns false, reject *promise* with a TypeError and abort these steps.
- Return the result of running map local type to NDEF given *record* and *ndef*.

4. Switching on *record*'s recordType, pass *record* and *ndef* to one of the following algorithms and return the result. If the algorithm throws an exception *e*, reject *promise* with *e* and abort these steps.

"empty"

- Return map empty record to NDEF given *record* and *ndef*.

"text"

- Return map text to NDEF given *record* and *ndef*.

"url"

- Return map a URL to NDEF given *record* and *ndef*.

"mime"

- Return map binary data to NDEF given *record* and *ndef*.

"smart-poster"

- Return map smart poster to NDEF given *record* and *ndef*.

"absolute-url"

- Return map absolute-URL to NDEF given *record* and *ndef*.

5. If running validate external type on *record*'s recordType returns true, return map external data to NDEF given *record* and *ndef*.

6. Otherwise, throw a TypeError and abort these steps.

9.10.2 Validating external type

The [NFC-RTD] specifies that external types *MUST* contain the domain name of the issuing organization, a colon U+003A (:) and a type name that is at least one

character long, for instance "w3.org:member".

The [NFC-RTD] specifies the URN prefix "urn:nfc:ext:" as well, but it is not stored in the NDEF record, therefore Web NFC applications *SHOULD NOT* specify the URN prefix when creating external type records.

NOTE

The [NFC-RTD] requires that external type names are represented with the URN prefix "urn:nfc:ext:", e.g. when reading NDEF messages. However, since external type records are distinguished by having the TNF FIELD set to 0x04, there is no risk seen for type name clashing. Also, there are W3C TAG recommendations to avoid using URNs in the Web. Therefore, Web NFC does not use the URN prefix neither when reading or writing NDEF messages.

To **validate external type** given *input*, run these steps:

1. If *input* is not an ASCII string, is empty, or its length exceeds 255 bytes, return **false**.
2. Let *domain* be the *input* from the start of *input* up to but excluding the first occurrence of U+003A (:), or **null** if that is not found.
3. Let *type* be the *input* after the first occurrence, if any, of U+003A (:) up to the end of *input*, or **null** if that is not found.
4. If *domain* or *type* is **null**, return **false**.
5. Let *asciiDomain* be the result of running domain to ASCII given *domain* and **true** (as *beStrict*).
6. If *asciiDomain* is failure, return **false**.
7. If *asciiDomain* contains a forbidden host code point or U+005F LOW LINE (_), return **false**.
8. If *type* contains code points that are not ASCII alphanumeric, or U+0024 (\$), U+0027 ('), U+0028 LEFT PARENTHESIS ((), U+0029 RIGHT PARENTHESIS ()), U+002A (*), U+002B



(+), U+002C (,), U+002D (-), U+002E (.), U+003B (;), U+003D (=), U+0040 (@), U+005F (_),

return **false**.

9. Return **true**.

9.10.3 Validating local type

To **validate local type** given an *input* run these steps:

1. Let *localTypeName* be the *input* after the first occurrence of U+003A (:) up to the end of *input*.
2. If *localTypeName* is not a USVString or its length exceeds 255 bytes, return **false** and abort these steps.
3. If *localTypeName* does not start with a lowercase character or a number, return **false**.
4. If *input* is equal to the record type of any NDEF record defined in its containing NDEF message, return **false**.
5. Return **true**.

9.10.4 Mapping empty record to NDEF

To **map empty record to NDEF** given a *record* and *ndef*, run these steps:

1. If *record*'s mediaType is not **undefined**, throw a TypeError and abort these steps.
2. If *record*'s id is not **undefined**, throw a TypeError and abort these steps.
3. Set the *ndef*'s TNF field to **0** (empty record).
4. Set the *ndef*'s IL field to **0**.
5. Set *ndef*'s TYPE LENGTH field, and PAYLOAD LENGTH field to **0**, and omit TYPE field and PAYLOAD field.
6. Return *ndef*.

9.10.5 Mapping string to NDEF

To **map text to NDEF** given a *record* and *ndef*, run these steps:

NOTE

This is useful when clients specifically want to write text in a [well-known type record](#). Other options would be to use the value `"mime"` with an explicit [MIME type](#) text type, which allows for better differentiation, e.g. when using `"text/xml"`, or `"text/vcard"`.

1. If *record*'s `mediaType` is not `undefined`, [throw](#) a `TypeError` and abort these steps.
2. If the type of *record*'s `data` is not `DOMString` or `BufferSource`, [throw](#) a `TypeError` and abort these steps.
3. Let *documentLanguage* be the [document element](#)'s `lang` attribute.
4. If *documentLanguage* is the empty string, set it to `"en"`.
5. Let *language* be *record*'s `lang` if it [exists](#), or else to *documentLanguage*.
6. Switch on the type of *record*'s `data`:

`DOMString`

1. If *record*'s `encoding` is neither `undefined` nor `"utf-8"`, [throw](#) a `TypeError` and abort these steps.
2. Let *encoding label* be `"utf-8"`.

`BufferSource`

1. Let *encoding label* be *record*'s `encoding` if it [exists](#), or else `"utf-8"`.
2. If *encoding label* is not equal to `"utf-8"`, `"utf-16"`, `"utf-16le"` or `"utf-16be"` [throw](#) a `TypeError` and abort these steps.

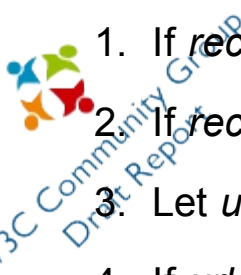
7. Let *encoding name* be the [name obtained](#) from *encoding label*.
8. Let *header* be a [byte](#) constructed the following way:
 1. If *encoding name* is equal to UTF-8, set bit 7 to the value 0, or else set the value to 1.
 2. Set bit 6 to the value 0 (reserved).



3. Let *languageLength* be the length of the *language* string.
 4. If *languageLength* cannot be stored in 6 bit (*languageLength* > 63), throw a *SyntaxError*.
 5. Set bit 5 to bit 0 to *languageLength*.
9. Let *data* be an empty byte sequence.
 1. Set the first byte (position 0) of *data* to *header*.
 2. Set position 1 (second byte) to position *languageLength* of *data* to *language*.
 3. Switch on the type of *record*'s data:
 - DOMString
 1. Let *stream* be the resulting byte stream of running UTF-8 encode on *record*'s data.
 2. Read bytes from *stream* into *data* (from position *languageLength* + 1) until read returns end-of-stream.
 - BufferSource
 1. Set bytes from *record*'s data into *data* (from position *languageLength* + 1) .
 10. Set *length* to the length of *data*.
 11.
 1. Set the *ndef*'s TNF field to 1 (well-known type record).
 2. Set the *ndef*'s TYPE field to "T" (0x54).
 3. Set the *ndef*'s PAYLOAD LENGTH field to *length*.
 4. If *length* > 0, set the *ndef*'s PAYLOAD field to *data*.
 12. Return *ndef*.

9.10.6 Mapping URL to NDEF

To *map a URL to NDEF* given a *record* and *ndef*, run these steps:

- 
1. If *record*'s *mediaType* is not *undefined*, *throw* a *TypeError* and abort these steps.
 2. If *record*'s *data* is not a *DOMString*, *throw* a *TypeError* and abort these steps.
 3. Let *url* be the result of *parsing* *record*'s *data*.
 4. If *url* is failure, *throw* a *TypeError* and abort these steps.
 5. Let *serializedURL* be *serialization* of *url*.
 6. Match the URI prefixes as defined in *NFC Forum Technical Specifications*, URI Record Type Definition specification, Section 3.2.2, against the *serializedURL*.
 7. Let *prefixString* be the matched prefix or else the *empty string*.
 8. Let *prefixByte* be the corresponding prefix number, or else 0.
 9. Let *shortenedURL* be *serializedURL* with *prefixString* removed from the start of the *string*.
 10. Let *data* be an empty *byte sequence*.
 1. Set the first *byte* of *data* to *prefixByte*.
 2. Let *stream* be the resulting *byte stream* of running *UTF-8 encode* on *shortenedURL*.
 3. *Read* bytes from *stream* into *data* (from position 1) until *read* returns *end-of-stream*.
 11. Set *length* to the *length* of *data*.
 12. Set the *ndef*'s *TNF field* to 1 (*well-known type record*).
 13. Set the *ndef*'s *TYPE field* to "U" (0x55).
 14. Set the *ndef*'s *PAYLOAD LENGTH field* to *length*.
 15. If *length* > 0, set the *ndef*'s *PAYLOAD field* to *data*.
 16. Return *ndef*.

9.10.7 Mapping binary data to NDEF

To *map binary data to NDEF* given a *record* and *ndef*, run these steps:



1. If the type of a *record*'s data is not BufferSource, throw a TypeError and abort these steps.
2. Let *mimeTypeRecord* be the MIME type returned by running parse a MIME type on *record*'s mediaType.
 1. If *mimeTypeRecord* is failure, let *mimeTypeRecord* be a new MIME type record whose type is "application", and subtype is "octet-stream".
3. Set *arrayBuffer* to *record*'s data.
4. Set *length* to *arrayBuffer*.[[ArrayBufferByteLength]].
5. Set *data* to *arrayBuffer*.[[ArrayBufferData]].
6. Set the *ndef*'s TNF field to 2 (MIME type).
7. Set the *ndef*'s TYPE field to the result of serialize a MIME type with *mimeTypeRecord* as the input.
8. Set the *ndef*'s PAYLOAD LENGTH field to *length*.
9. If *length* > 0, set the *ndef*'s PAYLOAD field to *data*.
10. Return *ndef*.

9.10.8 Mapping external data to NDEF

To **map external data to NDEF** given a *record* and *ndef*, run these steps:

1. If *record*'s mediaType is not undefined, throw a TypeError and abort these steps.
2. If the type of a *record*'s data is not BufferSource or NDEFMessageInit, throw a TypeError and abort these steps.
3. Set *ndef*'s TNF field to 4 (external type record).
4. Set the *ndef*'s TYPE field to *record*'s recordType.
5. If the type of a *record*'s data is BufferSource,
 1. Set *arrayBuffer* to *record*'s data.
 2. Set *length* to *arrayBuffer*.[[ArrayBufferByteLength]].
 3. Set *data* to *arrayBuffer*.[[ArrayBufferData]].



4. Set the *ndef*'s PAYLOAD LENGTH field to *length*.
5. If *length* > 0, set the *ndef*'s PAYLOAD field to *data*.
6. If the type of a *record*'s *data* is NDEFMessageInit,
 1. Set the *ndef*'s PAYLOAD field to the result of running the create NDEF message given *record*'s *data*.
 2. Set the *ndef*'s PAYLOAD LENGTH field to the length of *ndef*'s PAYLOAD field.
7. Return *ndef*.

9.10.9 Mapping local type to NDEF

To **map local type to NDEF** given a *record* and *ndef*, run these steps:

1. If *record*'s mediaType is not undefined, throw a TypeError and abort these steps.
2. If the type of a *record*'s *data* is not BufferSource or NDEFMessageInit, throw a TypeError and abort these steps.
3. Set *ndef*'s TNF field to 1 (well-known type record).
4. Let *localTypeName* be the *record*'s recordType after the first occurrence of U+003A (:) up to the end of *record*'s recordType.
5. Set *ndef*'s TYPE field to *localTypeName*, representing the local type name.
6. If the type of a *record*'s *data* is BufferSource,
 1. Set *arrayBuffer* to *record*'s *data*.
 2. Set *length* to *arrayBuffer*.[[ArrayBufferByteLength]].
 3. Set *data* to *arrayBuffer*.[[ArrayBufferData]].
 4. Set the *ndef*'s PAYLOAD LENGTH field to *length*.
 5. If *length* > 0, set the *ndef*'s PAYLOAD field to *data*.
7. If the type of a *record*'s *data* is NDEFMessageInit,
 1. Set the *ndef*'s PAYLOAD field to the result of running the create NDEF message given *record*'s *data*.



2. Set the *ndef*'s PAYLOAD LENGTH field to the length of *ndef*'s PAYLOAD field.

8. Return *ndef*.

9.10.10 Mapping smart poster to NDEF

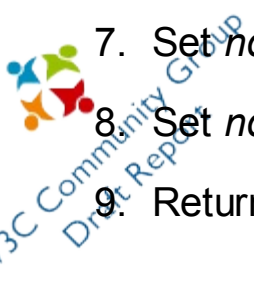
To **map smart poster to NDEF**, given a *record* and *ndef*, run these steps:

1. If *record*'s mediaType is not undefined, throw a TypeError and abort these steps.
2. If the type of a *record*'s data is not NDEFMessageInit, throw a TypeError and abort these steps.
3. Set *ndef*'s TNF field to **1** (well-known type record).
4. Set *ndef*'s TYPE field to "Sp" (0x53 0x70).
5. Set *ndef*'s PAYLOAD field to the result of running the create NDEF message given *record*'s data.
6. Set *ndef*'s PAYLOAD LENGTH field to the length of *ndef*'s PAYLOAD field.
7. Return *ndef*.

9.10.11 Mapping absolute-URL to NDEF

To **map absolute-URL to NDEF** given a *record* and *ndef*, run these steps:

1. If *record*'s mediaType is not undefined, throw a TypeError and abort these steps.
2. If *record*'s data is not a DOMString, throw a TypeError and abort these steps.
3. If the result of parsing *record*'s data is failure, throw a SyntaxError and abort these steps.
4. Set *arrayBuffer* to *record*'s data.
5. Set *data* to *arrayBuffer*.[[ArrayBufferData]].
6. Set *ndef*'s TNF field to **3** (absolute-URL record).



7. Set *ndef*'s [TYPE field](#) to *data*.

8. Set *ndef*'s [PAYLOAD LENGTH field](#) to 0 and omit [PAYLOAD field](#).

9. Return *ndef*.

9.11 Listening for content

To listen for [NFC content](#), the client *MUST* activate an [NDEFReader](#) instance by calling [NDEFReader.scan\(\)](#). When attaching an event listener for the "reading" event on it, [NFC content](#) is accessible to the client.

If there are any [NDEFReader](#) instances in [activated reader objects](#) then the [UA](#) *MUST* listen to [NDEF messages](#) on all connected NFC adapters.

Each [NDEFReader](#) can accept [NDEF messages](#) based on data type, and record identifier filters.

9.11.1 Match patterns

A *match pattern* is defined by the following ABNF:

```
match-pattern = top-level-type "/" [ tree "." ] subtype [ "+" suffix ] [ ";" parameter s ]
top-level-type = "*" / < VCHAR except "/" and "*" >
subtype        = "*" / < VCHAR except "+" >
```

A [match pattern](#) is a [glob](#) used for matching [MIME types](#), for instance the pattern "application/*+json" matches "application/calendar+json", but does not match "application/json". The pattern "/*+json", on the other hand, matches both.

9.11.2 The scan() method

Incoming [NFC content](#) is matched using [NDEFReader](#) instances.

When the `NDEFReader.scan` method is invoked, the UA *MUST* run the following **NFC listen algorithm**:

1. Let *p* be a new `Promise` object.
2. Let *reader* be the `NDEFReader` instance.
3. Let *options* be first argument.
4. **For each** *key* → *value* of *options*:
 1. If *key* equals `"signal"` and *value* is not `undefined`, set *reader*.`[[Signal]]` to *value*.
 2. Otherwise, if *key* equals `"id"`, set *reader*.`[[Id]]` to *value*.
 3. Otherwise, if *key* equals `"recordType"`, set *reader*.`[[RecordType]]` to *value*.
 4. Otherwise, if *key* equals `"mediaType"`, set *reader*.`[[MediaType]]` to *value*.
5. If *reader*.`[[Signal]]`'s `aborted flag` is set, then reject *p* with a `"AbortError"` `DOMException` and return *p*.
6. If *reader*.`[[Signal]]` is not `null`, then **add the following abort steps** to *reader*.`[[Signal]]`:
 1. Remove the `NDEFReader` instance from the `activated reader objects`.
 2. If the `activated reader objects` is empty, then make a request to stop listening to `NDEF messages` on all `NFC adapters`.
7. Return *p* and run the following steps **in parallel**:
 1. If the `obtain permission` steps return `false`, then reject *p* with a `"NotAllowedError"` `DOMException` and abort these steps.
 2. If there is no underlying `NFC Adapter`, or if a connection cannot be established, then reject *p* with a `"NotSupportedError"` `DOMException` and abort these steps.
 3. If the UA is not allowed to access the underlying `NFC Adapter` (e.g. a user preference), then reject *p* with a `"NotReadableError"` `DOMException` and abort these steps.
 4. Add *reader* to the `activated reader objects`.

5. Resolve p .

6. Whenever the [UA](#) detects NFC technology, run the [NFC reading algorithm](#).

9.11.3 The NFC reading algorithm

To receive [NDEF](#) content, run the ***NFC reading algorithm***:

1. If [NFC is suspended](#), abort these steps.
2. If there is a [pending write tuple](#) and its writer's option's ignoreRead is **true**, abort these steps.
3. If the [NFC tag](#) in proximity range does not expose [NDEF](#) technology for reading or formatting, run the following sub-steps:
 1. **For each** [NDEFReader](#) instance *reader* in the [activated reader objects](#), run the following sub-steps:
 1. **Fire an event** named "**error**" at *reader*.
 2. Abort these steps.
4. Let *serialNumber* be the device identifier as a series of numbers, or **null** if unavailable.
5. If *serialNumber* is not **null**, set it to the [string](#) of U+003A (:) concatenating each number represented as [ASCII hex digit](#), in the same order.
6. Let *message* be a new [NDEFMessage](#) object, with *message*'s records set to the empty [list](#).
7. If the [NFC tag](#) in proximity range is unformatted and is NDEF-formattable, let *input* be **null**. Otherwise, let *input* be the notation for the [NDEF message](#) which has been received.



NOTE

The UA *SHOULD* represent an unformatted NFC tag as an NDEF message containing no NDEF records, i.e. an empty array for its records property.

8. For each NDEF record which is part of *input*, run the following sub-steps:
 1. Let *ndef* be the notation for the current NDEF record with *typeNameField* corresponding to the TNF field and *payload* corresponding to the PAYLOAD field data.
 2. Let *record* be the result of parse an NDEF record on *ndef*.
 3. If *record* is not **null**, append *record* to *message*'s records.
9. If NFC is not suspended, run the dispatch NFC content steps with given *serialNumber* and *message*.

9.11.4 Dispatching NFC content

To **dispatch NFC content** given a *serialNumber* of type serialNumber and a *message* of type NDEFMessage, run these steps:

1. For each NDEFReader instance *reader* in the activated reader objects, run the following sub-steps:
 1. For each *record* in *message*,
 1. Let *matched* be **false**.
 2. If *reader*.[[Id]] is not **undefined** and is not equal to *record*'s id, continue. If it is equal, set *matched* to **true**.
 3. If *reader*.[[RecordType]] is not **undefined** and is not equal to *record*'s recordType, continue. If it is equal, set *matched* to **true**.
 4. If *reader*.[[MediaType]] is not **undefined** and is not equal to *record*'s mediaType, continue. If it is equal, set *matched* to **true**.

5. If *matched* is **true**,

1. fire an event named "**reading**" at *reader* using NDEFReadingEvent with its serialNumber attribute initialized to *serialNumber* and message attribute initialized to *message*.
2. Break.

9.12 Parsing content

9.12.1 Parsing records from bytes

To ***parse records from bytes*** given *bytes*, run these steps:

1. If the length of *bytes* is **0**, return **null** and abort these steps.
2. Let *records* be the empty list.
3. As long as there are unread bytes of *bytes*, run the following sub-steps:
 1. If the remaining length of *bytes* is less than **3**, return **null** and abort these sub-steps.
 2. If any of the following steps requires reading bytes beyond the remaining length of *bytes*, return **null** and abort these sub-steps.
 3. Let *ndef* be the notation for the current NDEF record.
 4. Let *header* be the next byte of *bytes*.
 1. Let *messageBegin* (MB field) be the left most bit (bit 7) of *header*.
 2. If this is the first iteration of these sub-steps and *messageBegin* is **false**, return **null** and abort these sub-steps.
 3. Let *messageEnd* (ME field) be bit 6 of *header*.

NOTE

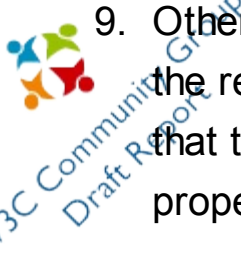
As chunked records are not allowed as sub records, ignore bit 5 ([CF field](#)) is ignored.

4. Let *shortRecord* ([SR field](#)) be bit 4 of *header*.
 5. Let *hasIdLength* ([IL field](#)) be bit 3 of *header*.
 6. Let *ndef*'s *typeNameField* ([TNF field](#)) be the integer value of bit 2-0 of *header*.
 5. Let *typeLength* be the integer value of next byte ([TYPE LENGTH field](#)) of *bytes*.
 6. If *shortRecord* is **true**, let *payloadLength* be the integer value of next byte ([PAYLOAD LENGTH field](#)) of *bytes*.
 7. Otherwise, let *payloadLength* be the integer value of the next 4 bytes of *bytes*.
 8. If *hasIdLength* is **true**, let *idLength* be the integer value of next byte ([ID LENGTH field](#)) of *bytes*, otherwise let it be **0**.
 9. If *typeLength* > 0, let *ndef*'s *type* be result of running [UTF-8 decode](#) on the next *typeLength* ([TYPE field](#)) bytes, or else let *type* be the empty string.
 10. If *idLength* > 0, let *ndef*'s *id* be result of running [UTF-8 decode](#) on the next *idLength* ([ID field](#)) bytes, or else let *ndef*'s *id* be the empty string.
 11. Let *ndef*'s *payload* be the [byte sequence](#) of the last *payloadLength* ([PAYLOAD field](#)) bytes, which may be **0** bytes.
 12. Let *record* be the result of [parse an NDEF record](#) on *ndef*.
 13. If *record* is not **null**, [append](#) *record* to *records*.
 14. If *messageEnd* is **true**, abort these sub-steps.
4. Return *records*.

9.12.2 Parsing NDEF records

To **parse an NDEF record** given *ndef* into a *record*, run these steps:

1. Set *record*'s id to *ndef*'s *id*.
2. Set *record*'s lang to *null*.
3. Set *record*'s encoding to *null*.
4. If *ndef*'s *typeNameField* (TNF field) is 0 (empty record):
 1. Set *record*'s id to *null*.
 2. Set *record*'s recordType to "empty".
 3. Set *record*'s mediaType to *null*.
 4. Set *record*'s data to *null*.
5. If *ndef*'s *typeNameField* is 1 (well-known type record):
 1. Set *record* to the result of the algorithm below switching on *ndef*'s *type*:
 - "T" (0x54)
 - Running parse an NDEF text record on *ndef*.
 - "U" (0x55)
 - Running parse an NDEF URL record on *ndef*.
 - "Sp" (0x53 0x70)
 - Running parse an NDEF smart-poster record on *ndef*.
6. If *ndef*'s *typeNameField* is 2 (MIME type record), then set *record* to the result of running parse an NDEF MIME type record on *ndef*, or make sure that the underlying platform provides equivalent values to the *record* object's properties.
7. Otherwise, if *ndef*'s *typeNameField* is 3 (absolute-URL record), then set *record* to the result of running parse an NDEF absolute-URL record on *ndef*.
8. Otherwise, if *ndef*'s *typeNameField* is 4 (external type record), then set *record* to the result of running parse an NDEF external type record on *ndef*, or make sure that the underlying platform provides equivalent values to the *record* object's properties.

- 
9. Otherwise, if *ndef*'s *typeNameField* is 5 ([unknown record](#)) then set *record* to the result of running [parse an NDEF unknown record](#) on *ndef*, or make sure that the underlying platform provides equivalent values to the *record* object's properties.

9.12.3 Parsing NDEF well-known **T** records

To **parse an NDEF text record** given an *ndefRecord* into a *record*, run these steps:

1. Set *record*'s *recordType* to "text".
2. Set *record*'s *mediaType* to null.
3. If *ndefRecord*'s [PAYLOAD field](#) is not present, set *record*'s *data* to null and return *record*.
4. Let *header* be the first [byte](#) of *ndefRecord*'s [PAYLOAD field](#).
5. Let *languageLength* be the value given by bit 5 to bit 0 of the *header*.
6. Let *language* be the result of running [ASCII decode](#) on second [byte](#) to the *languageLength* + 1 byte, inclusive.
7. Set *record*'s *lang* to *language*.
8. Set *record*'s *encoding* be "utf-8" if bit 7 ([MB field](#)) of *header* is equal to the value 0, or else "utf-16be".
9. Let *buffer* be the [byte sequence](#) of *ndefRecords*'s [PAYLOAD field](#).
10. Set *record*'s *data* to *buffer*.
11. Return *record*.



NOTE

The Unicode standard defines multiple encodings like UTF-8, UTF-16 and UTF-32. UTF-8 is the preferred encoding on the web, and it has the advantage that it is endianness agnostic, as [code points](#) are represented in single bytes.

The NDEF text records allow the text to be encoded as either UTF-8 or UTF-16. Generally, it is preferred to use UTF-8 encoding on the web, but integration with existing systems may require UTF-16.

The data transmission order and thus the byte order of NDEF is defined as big endian (BE) in [NFC Data Exchange Format \(NDEF\) Technical Specification](#), which means that everything is read back with big endian byte ordering.

For UTF-16, byte order matters as it might differ between reading and writing due to each [code point](#) spanning two bytes. For this reason UTF-16 encoded text usually contains a byte order mark (also known as BOM), which is written as `0xFEFF`. This means that if the byte order differs between host machine and NDEF, then the value will be read back as `0xFFFE`, indicating that the byte order should be swapped. The [Encoding Standard](#) differentiates UTF-16 as UTF-16BE (big endian) and UTF-16LE (little endian) depending on the byte order.

In the case that no byte order mark is present, UTF-16BE encoding should be assumed.

Using the [decoder](#) with `encoding` set to `utf-16`, it will automatically detect whether bytes should be swapped in case the byte order mark is present. `utf-16be` can be used to read as big endian in case there is no byte order mark.

Using the [encoder](#), it is only possible to encode as UTF-8, so if UTF-16 is needed, it will have to be encoded manually or by using a library.

9.12.4 Parsing NDEF well-known `u` records

To **parse an NDEF URL record** given an *ndefRecord* into a *record*, run these steps:

1. Set *record*'s recordType to "url".
2. Set *record*'s mediaType to null.
3. If *ndefRecord*'s PAYLOAD field is not present, set *record*'s data to null and return *record*.
4. Let *buffer* be the byte sequence of *ndefRecords*'s PAYLOAD field.
5. Let *prefixByte* be the value of the first byte of *buffer*.
6. If the value of *prefixByte* matches the URL expansion codes in the NFC Forum Technical Specifications URI Record Type Definition specification, Section 3.2.2, Table 3, then
 1. Let *prefixString* be the byte sequence value corresponding to the value of *prefixByte*.
 2. Set *record*'s data to *prefixString* appended to *buffer*.
7. Otherwise, if there is no match for *prefixByte*, set *record*'s data to *buffer*.
8. Return *record*.

9.12.5 Parsing NDEF well-known **sp** records

To **parse an NDEF smart-poster record** given an *ndefRecord* into a *record*, run these steps:

1. Set *record*'s recordType to "smart-poster".
2. Set *record*'s mediaType to null.
3. If *ndefRecord*'s PAYLOAD field is not present, set *record*'s data to null and return *record*.
4. Let *buffer* be the byte sequence of *ndefRecords*'s PAYLOAD field.
5. Set *record*'s data to *buffer*.
6. Return *record*.

9.12.6 Parsing NDEF MIME type records

To **parse an NDEF MIME type record** given an *ndefRecord* into a *record*, run these steps:

1. Set *record*'s recordType to "mime".
2. Set *record*'s mediaType to the result of serialize a MIME type with *mimeType* as the input.
3. Let *buffer* be the byte sequence of *ndefRecords*'s PAYLOAD field if that exists, or otherwise *null*.
4. Set *record*'s data to *buffer*.
5. Return *record*.

9.12.7 Parsing NDEF absolute-URL records

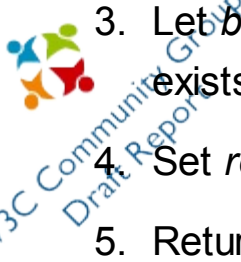
To **parse an NDEF absolute-URL record** given an *ndefRecord* into a *record*, run these steps:

1. Set *record*'s recordType to "absolute-url".
2. Set *record*'s mediaType to *null*.
3. Let *buffer* be the byte sequence of *ndefRecords*'s TYPE field.
4. Set *record*'s data to *buffer*.
5. Return *record*.

9.12.8 Parsing NDEF external type records

To **parse an NDEF external type record** given an *ndefRecord* into a *record*, run these steps:

1. Set *record*'s recordType to the value of *ndefRecord*'s TYPE field.
2. Set *record*'s mediaType to *null*.

- 
3. Let *buffer* be the byte sequence of *ndefRecords*'s PAYLOAD field if that exists, or otherwise `null`.
 4. Set *record*'s data to *buffer*.
 5. Return *record*.

9.12.9 Parsing NDEF unknown type records

To ***parse an NDEF unknown record*** given an *ndefRecord* into a *record*, run these steps:

1. Set *record*'s recordType to `"unknown"`.
2. Set *record*'s mediaType to `null`.
3. Let *buffer* be the byte sequence of *ndefRecords*'s PAYLOAD field if that exists, or otherwise `null`.
4. Set *record*'s data to *buffer*.
5. Return *record*.

10.1 Chain of trust

Web sites and applications using Web NFC are not trusted. This means that the user needs to be made aware of what NFC functionality a web page intends to do. Implementations need to make sure that when the user authorizes a method of this API, only that action is run, without side effects, and exactly in the context and the number of times the user allows the execution of NFC related operations, according to the algorithmic steps detailed in this specification.

Web NFC does not sign [NFC content](#). In order to protect the integrity and authenticity of NDEF messages, the NFC Forum introduced [[NDEF-SIGNATURE](#)]. Using [NDEF signature](#) and key management is the responsibility of the application.

For trusting the **confidentiality** of the data exchanged via NFC, applications may use encrypted [NFC content](#).

For trusting the **integrity** of the data exchanged via NFC, applications may use an [NDEF signature](#), with key management based on Public Key Infrastructure (PKI).

Security considerations for MIME types in general are discussed in [[RFC2048](#)] and [[RFC2046](#)].

10.2 Assets

This section is non-normative.

Assets to be protected include the following:

- **NDEF message** as a whole, **NDEF records** (including payload and header) in particular, either in-transfer or in-storage state, when they are being overwritten by a Web NFC triggered operation, against data disclosure and data modification. This also includes Denial of Service attacks against a



solution deployed with NFC tags (e.g. a malicious actor destroying tags linked to a solution).

- **User identity or other privacy sensitive attributes** that can be directly or indirectly determined by using Web NFC, by the [NFC content](#) creator, or by a web site using Web NFC. This data could be used directly or leaked forward to third parties. Examples are user location, device identifiers and user identifiers.
- **User data** exposed to a web page using Web NFC. While a web page might collect user data using other means than Web NFC, it might embed this data into NDEF records and share via Web NFC.
- **Integrity of user device.** A read of an NFC tag might result in a user device compromise that can further lead to loss of other web NFC or platform assets.

10.3 Attacker model

This section is non-normative.

The following attacker patterns have been considered:

- **Malicious web page creator:** phishing user data, identity or other privacy sensitive attributes, destroying or modifying NFC tags to cause further damage through fake identities and attack vectors.
- **Malicious NFC tag creator:** same as malicious web page owner, but it has a possibility to create, delete or modify the NFC tags locally. As a result can compromise integrity of user device, cause data injection, redirect to malicious web page, phishing user location, causing side actions such as installing applications, trigger automated dispatching or other actions.
- **Malicious man-in-the-middle (MITM) user:** any MITM style attack between Web NFC implementation and an [NFC adapter](#) in a user device, including attempts to interact with a web site using Web NFC by presenting modified or replayed NDEF records.

This section is non-normative.

An introduction to NFC security is found [here](#). Potential threats for Web NFC are given below.

10.4.1 Fingerprinting and data collection

Threat description

Malicious web page collects user data, identity or other privacy sensitive attributes (such as location) without user consent and exposes it to third parties (writing it to NFC tags).

Affected assets

User data, user identity or other privacy sensitive attributes

Actors

Malicious web page owner using Web NFC, malicious tag owner.

Mitigation, comments

The user *SHOULD* be able to be aware of what data can be shared using NFC from the given web page. Use permissions and user prompts for accessing personal data, minimize user data exposed to NFC. An NFC tag *SHOULD NOT* ever trigger a user's device to navigate to a web site without asking permission, unless the site has been in the foreground or has been brought to the foreground and has been granted permission. User agents *SHOULD* take into account the security and privacy measures listed in the [Geolocation API](#).

10.4.2 NFC tag modification

Threat description

An NFC tag is being modified without user consent. This might enable further attacks using a malicious tag or can be a Denial of Service attack to make one or more tags unusable.



Affected assets

NDEF message records, including payload and header in-storage.

Actors

Malicious web page creator, malicious user.

Mitigation, comments

Require permission and user prompt needed for writing tags. Or, control what tags can be written by a given web page, for instance a web page can write only a tag that can be connected to its [origin](#). Or, allow overwriting since tags not meant to be written can be protected by making them read only. Use [NDEF signature](#) to detect a modification of NFC tags.

10.4.3 NDEF record modification in-transit

Threat description

[NDEF records](#) transferred between Web NFC and the [NFC adapter](#) and user device are modified to cause various man-in-the-middle attacks or denial-of-service (DoS) attacks. Also, [NDEF signature](#) records can be removed or replaced along with changed content.

Affected assets

[NDEF records](#) in-transfer.

Actors

Malicious man-in-the-middle user.

Mitigation, comments

This threat is out of scope for Web NFC implementations. Applications can use [NDEF signatures](#) and appropriate tools (signature algorithm, certificates, security policies) to protect the [NFC content](#). Additionally, harden the platform stack.

10.4.4 NDEF record payload disclosure

Threat description

Confidential payload of [NDEF record](#) in-storage (stored on an NFC tag) or in-transfer between Web NFC and the [NFC adapter](#) are read by unauthorized



parties.

Affected assets

Confidential NDEF message payload in-transfer and in-storage.

Actors

Malicious man-in-the-middle user, malicious web page creator.

Mitigation, comments

To ensure confidentiality, use payload encryption and secure communication for data exchange, authentication and authorization between Web NFC and [NFC adapters](#).

10.4.5 Active attack via malicious NFC tag

Threat description

Malicious tag may be involuntarily or voluntarily read by devices and the data read may constitute an attack vector on the user agent. For example it can attempt to trigger an action on the device, which may be a threat, for instance launching a malicious web site, or opening an image prepared for attacking the device.

Affected assets

Integrity of user device, all other Web NFC assets.

Actors

Malicious tag creator.

Mitigation, comments

This is a generic problem with all existing NFC tags. The data is considered application specific. Implementations need security hardening. Involuntary touch is low probability due to short range and critical angle for reading, and due to the focus requirements. Automatic actions for smart posters and other tags should not be allowed. The user must be made aware and given the ability to control what is happening during the NFC communication. For instance, opening content from [smart poster](#), automatic connection to (possibly malicious) WiFi via [NFC handover](#), etc. Do not allow actions from untrusted NFC tags, trust can be established via the [NDEF signature](#) check.



10.5 Security mechanisms for implementations

This section is non-normative.

10.5.1 Obtaining permission

Implementations *SHOULD* use a mechanism to [obtain permission](#), for instance an explicit permission given by the user. The [Permissions](#) API is suggested to be used by UAs for implementing NFC related permissions.

Implementations *MAY* use per-session/ephemeral permissions.

10.5.2 Warning user during NFC operations

Implementations *MAY* show an overlay dialog whenever the NFC adapter is being accessed by the web page (e.g. there is an ongoing scan) in order to warn user.

10.6 Security mechanisms for applications

This section is non-normative.

10.6.1 Encrypting [NFC content](#)

For trusting the confidentiality of the data exchanged via NFC, applications may use encrypted [NFC content](#) with key management based on Public Key Infrastructure (PKI). Key management is out of the scope of Web NFC.

10.6.2 Signing NDEF records

For trusting the integrity of the data exchanged via NFC, user agents *MAY* use an [NDEF signature](#) with a Public Key Infrastructure for key management.

For tags signed with [NDEF signature](#) version 1.0 ([\[NFC-SECURITY\]](#)), the signature is applied only to the [TYPE field](#), [ID field](#) and [PAYLOAD field](#), leaving out the first byte of the NDEF header, allowing surface to attacks. Version 2.0 of [\[NFC-SECURITY\]](#) included tag hardware attributes in the signature and allowed for shorter certificates.

An [NDEF signature](#) covers the preceding records until another [NDEF signature](#) or the beginning of the [NDEF message](#) is reached.

In order to mitigate [known vulnerabilities](#) of [NDEF signature](#), it is recommended that applications always sign a full [NDEF message](#) with a single [NDEF signature](#), and use the right tool chain and security policies for creating and verifying signatures.

10.7 Security policies

This section lists the normative security policies for implementations.

10.7.1 Secure Context

Only [secure contexts](#) are allowed to access [NFC content](#). Browsers *MAY* ignore this rule for development purposes only.

10.7.2 Visible document

Web NFC functionality is allowed only for the [Document](#) of the [top-level browsing context](#), which must be [visible](#).

This also means that UAs should block access to the NFC radio if the display is off or the device is locked. For backgrounded web pages, receiving and writing [NFC content](#) must be [suspended](#).

10.7.3 Permissions controls

Making an [NFC tag](#) read-only *MUST* [obtain permission](#), or otherwise fail.

Setting up listeners for reading [NFC content](#) *SHOULD* [obtain permission](#).

Writing [NFC content](#) to an [NFC tag](#) *MUST* [obtain permission](#). See the [§ 9.9 Writing content](#) section.

All permission that are preserved beyond the current browsing session *MUST* be revocable.

10.7.4 Warn about risk of physical location leak

When listening for and writing [NFC content](#), the UA *MAY* warn the user that the given [origin](#) may be able to infer physical location.

10.7.5 Restrict automatic handling

When the payload data on [NFC content](#) is untrusted, it *MUST NOT* be used by the UA to do automatic handling of the content, such as opening a web page with a URL found in an [NFC tag](#), or installing an application, or other actions, unless the user approves that.

10.7.6 Signing [NFC content](#)

The following policies are recommended to be implemented by applications.

- A [smart poster](#) *MAY* be trusted only if signed by using a single [NDEF signature](#) record by the same issuer and it is either the first record in a message, or it is preceded by another [NDEF signature](#).
- An [NDEF message](#) *MAY* be trusted only if signed by a single [NDEF signature](#) record by the same issuer.



- User agents expose [NDEF signature](#) records without verifying them. It is applications' responsibility to verify signatures and to sign [NFC content](#).
- Applications *SHOULD* use appropriate signature algorithms, certificates and security policies for [NDEF signature](#) creation and verification. Also, take into account the known attacks against [NDEF signatures](#), for instance removing [NDEF signature](#), replacing [NDEF signature](#) along with modifying the [NFC content](#), reordering records by changing the record [PAYLOAD LENGTH field](#), etc.

WebIDL

```
[SecureContext, Exposed=Window]
interface NDEFMessage {
    constructor(NDEFMessageInit messageInit);
    readonly attribute FrozenArray<NDEFRecord> records;
};

dictionary NDEFMessageInit {
    required sequence<NDEFRecordInit> records;
};

typedef (DOMString or BufferSource or NDEFMessageInit) NDEFRecordDataSource;

[SecureContext, Exposed=Window]
interface NDEFRecord {
    constructor(NDEFRecordInit recordInit);

    readonly attribute USVString recordType;
    readonly attribute USVString? mediaType;
    readonly attribute USVString? id;
    readonly attribute DataView? data;

    readonly attribute USVString? encoding;
    readonly attribute USVString? lang;

    sequence<NDEFRecord>? toRecords();
};

dictionary NDEFRecordInit {
    required USVString recordType;
    USVString mediaType;
    USVString id;

    USVString encoding;
    USVString lang;

    NDEFRecordDataSource data;
};

typedef (DOMString or BufferSource or NDEFMessageInit) NDEFMessageSource;

[SecureContext, Exposed=Window]
interface NDEFWriter {
```



constructor();

Promise<void> write(NDEFMessageSource message, optional NDEFWriteOptions options={})
);
};

[SecureContext, Exposed=Window]
interface **NDEFReader** : EventTarget {
 constructor();

 attribute EventHandler onerror;
 attribute EventHandler onreading;

Promise<void> scan(optional NDEFScanOptions options={});
};

[SecureContext, Exposed=Window]
interface **NDEFReadingEvent** : Event {
 constructor(DOMString type, NDEFReadingEventInit readingEventInitDict);

 readonly attribute DOMString serialNumber;
 [SameObject] readonly attribute NDEFMessage message;
};

dictionary **NDEFReadingEventInit** : EventInit {
 DOMString? serialNumber = "";
 required NDEFMessageInit message;
};

dictionary **NDEFWriteOptions** {
 boolean ignoreRead = true;
 boolean overwrite = true;
 AbortSignal? signal;
};

dictionary **NDEFScanOptions** {
 USVString id;
 USVString recordType;
 USVString mediaType;
 AbortSignal? signal;
};



B. Acknowledgments

The editors would like to thank Jeffrey Yasskin, Anne van Kesteren, Anssi Kostiainen, Domenic Denicola, Daniel Ehrenberg, Jonas Sicking, Don Coleman, Salvatore Iovene, Rijubrata Bhaumik, Wanming Lin, Han Leon, and Ryan Sleevi for their contributions to this document.

Special thanks to Luc Yriarte and Samuel Ortiz for their initial [work](#) on exposing NFC to the web platform, and for their support for the current approach. Also, special thanks to Elena Reshetova for the contributions to the Security and Privacy section.

C.1 Normative references

[BCP47]

Tags for Identifying Languages. A. Phillips; M. Davis. IETF. September 2009. IETF Best Current Practice. URL: <https://tools.ietf.org/html/bcp47>

[dom]

DOM Standard. Anne van Kesteren. WHATWG. Living Standard. URL: <https://dom.spec.whatwg.org/>

[ECMAScript]

ECMAScript Language Specification. Ecma International. URL: <https://tc39.es/ecma262/>

[encoding]

Encoding Standard. Anne van Kesteren. WHATWG. Living Standard. URL: <https://encoding.spec.whatwg.org/>

[HTML]

HTML Standard. Anne van Kesteren; Domenic Denicola; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: <https://html.spec.whatwg.org/multipage/>

[infra]

Infra Standard. Anne van Kesteren; Domenic Denicola. WHATWG. Living Standard. URL: <https://infra.spec.whatwg.org/>

[mimesniff]

MIME Sniffing Standard. Gordon P. Hemsley. WHATWG. Living Standard. URL: <https://mimesniff.spec.whatwg.org/>

[NDEF-SIGNATURE]

NFC Forum Signature Record Type Definition. NFC Forum. 18 November 2010. URL: http://members.nfc-forum.org/specs/spec_list/

[NFC-NDEF]

NFC Data Exchange Format (NDEF) Technical Specification. NFC Forum. 24 July 2006. URL: http://members.nfc-forum.org/specs/spec_list/

[NFC-RTD]



[NFC-STANDARDS]

NFC Forum Technical Specifications. NFC Forum. 24 July 2006. URL: http://members.nfc-forum.org/specs/spec_list/

[PAGE-VISIBILITY]

Page Visibility (Second Edition). Jatinder Mann; Arvind Jain. W3C. 29 October 2013. W3C Recommendation. URL: <https://www.w3.org/TR/page-visibility/>

[PERMISSIONS]

Permissions. Mounir Lamouri; Marcos Caceres; Jeffrey Yasskin. W3C. 25 September 2017. W3C Working Draft. URL: <https://www.w3.org/TR/permissions/>

[RFC2046]

Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. N. Freed; N. Borenstein. IETF. November 1996. Draft Standard. URL: <https://tools.ietf.org/html/rfc2046>

[RFC2048]

Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. N. Freed; J. Klensin; J. Postel. IETF. November 1996. Best Current Practice. URL: <https://tools.ietf.org/html/rfc2048>

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: <https://tools.ietf.org/html/rfc2119>

[RFC5234]

Augmented BNF for Syntax Specifications: ABNF. D. Crocker, Ed.; P. Overell. IETF. January 2008. Internet Standard. URL: <https://tools.ietf.org/html/rfc5234>

[RFC5646]

Tags for Identifying Languages. A. Phillips, Ed.; M. Davis, Ed.. IETF. September 2009. Best Current Practice. URL: <https://tools.ietf.org/html/rfc5646>



[RFC8174]

Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba.

IETF. May 2017. Best Current Practice. URL:

<https://tools.ietf.org/html/rfc8174>

[secure-contexts]

Secure Contexts. Mike West. W3C. 15 September 2016. W3C Candidate

Recommendation. URL: <https://www.w3.org/TR/secure-contexts/>

[url]

URL Standard. Anne van Kesteren. WHATWG. Living Standard. URL:

<https://url.spec.whatwg.org/>

[WebIDL]

Web IDL. Boris Zbarsky. W3C. 15 December 2016. W3C Editor's Draft.

URL: <https://heycam.github.io/webidl/>

C.2 Informative references

[NDEF-SMARTPOSTER]

NFC Forum Smart Poster Record Type Definition. NFC Forum. 24 July 2006.

URL: http://members.nfc-forum.org/specs/spec_list/

[NDEF-TEXT]

NFC Forum Text Record Type Definition. NFC Forum. 14 August 2013. URL:

http://members.nfc-forum.org/specs/spec_list/

[NDEF-URI]

NFC Forum URI Record Type Definition. NFC Forum. 24 July 2006. URL:

http://members.nfc-forum.org/specs/spec_list/

[NFC-HANDOVER]

NFC Forum Connection Handover Technical Specification. NFC Forum. 16

January 2014. URL: http://members.nfc-forum.org/specs/spec_list/

[NFC-SECURITY]

Web NFC Security and Privacy. W3C. 25 April 2015. URL:

<https://github.com/w3c/web-nfc/security-privacy.html>

[RFC3986]



Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee; R.

Fielding; L. Masinter. IETF. January 2005. Internet Standard. URL:

<https://tools.ietf.org/html/rfc3986>

[RFC3987]

Internationalized Resource Identifiers (IRIs). M. Duerst; M. Suignard. IETF.

January 2005. Proposed Standard. URL: <https://tools.ietf.org/html/rfc3987>