life.augmented

AN5616

Application note

NDEF library

Introduction

The NFC data exchange format (NDEF) is used to transfer semantic content between devices. It is used especially by NFC-enabled iPhone® or Android™ smartphones to read and interpret NDEF messages from different NFC Forum tag types.

NDEF, among other things, allows user to open a web page, import contacts, pair devices (via Wi-Fi® or Bluetooth®).

The NDEF library is part of the ST25 NFC embedded library package (STSW-ST25R-LIB).

The STSW-ST25R-LIB embedded software provides middleware and their associated examples, which can be reused when developing an application with ST25R products.

The sample application contained within ST25 embedded NFC library includes also a demonstration for the NDEF library. The provided sample applications are built on top of RFAL. The NDEF library provides several functionalities required for NDEF message management.

See the NDEF specification for the message encapsulation format used to exchange information between two NFC Forum devices.

The reference section provides the technical specifications of the different types supported.

1 Application overview

The sample applications are available for the ST25R reader devices. These applications typically run on a NUCLEO-L476RG board, which has plugged in one of the X-NUCLEO-NFC03A1, X-NUCLEO-NFC05A1 or X-NUCLEO-NFC06A1 shields.

The applications currently provided are:

- NDEF read/write demonstration
- Card emulation and Bluetooth® pairing

Table 1. List of acronyms

Acronym	Description
NFC	Near field communication
RFAL	RF abstract layer
SDK	Software development kit
NDEF	NFC data exchange format
RTD	Record type definition
T2T	Type 2 tag
T3T	Type 3 tag
T4AT	Type 4A tag
T4BT	Type 4B tag
T5T	Type 5 tag
URI	Uniform resource identifier

Table 2. Version reference

Reference	Description
NDEF	NFC data exchange format, technical specification, version 1.0, 2017-11-18; NFC Forum
RTD_DI	Device information record type definition, technical specification, version 1.0, 2017-12-31, NFC Forum
RTD_TEXT	Text record type definition, technical specification, version 1.0, 2017-12-31, NFC Forum
RTD_URI	Record type definition, technical specification, version 1.0, 2017-12-31, NFC Forum
WLC	Wireless charging, technical specification, version 1.0, 2020-03-31, NFC Forum
BT	Bluetooth® secure simple pairing using NFC, application document, NFCForum-AD-BTSSP_1_1, 2014-01-09, NFC Forum
CH	Connection handover, technical specification, version 1.4, 2018-04-30, NFC Forum
WI-FI	Wi-Fi® simple configuration, technical specification, version 2.0.5, © 2014 Wi-Fi Alliance Wi-Fi® protected setup specification, version 1.0h, December 2006

2 Setup

For more information visit the STSW-ST25-LIB webpage available on www.st.com.

The sample applications are available for X-NUCLEO-NFC03, X-NUCLEO-NFC05 and X-NUCLEO-NFC06 boards. For details about the board function refer to the documentation of X-NUCLEO-NFC03A1, X-NUCLEO-NFC05A1 or X- NUCLEO-NFC06A1 and the documentation of X-CUBE-NFC3, X-CUBE-NFC5 or X-CUBE-NFC6.

2.1 Hardware setup

The following hardware components are needed:

- STM32 Nucleo development platform for STM32L476: NUCLEO-L476RG
- X-NUCLEO-NFC03A1, or X-NUCLEO-NFC05A1, or X- NUCLEO-NFC06A1
- One USB Type-A to USB mini-B cable to connect the STM32 Nucleo to the PC

2.2 Software setup

This section lists the minimum requirements for the developer to set up the SDK.

- Development tool-chains and compilers (STM32CubeIDE is the reference IDE for the ST25 NFC embedded library package)
- A serial terminal program (e. g. TeraTerm, HyperTerminal, Hterm) set to 115200 8n1
- Drivers for ST-Link virtual COM port. The Microsoft® Windows® 10 system includes the drivers. For other systems, check the STSW-LINK009

2.3 Running the application

The user locates the appropriate sample applications and opens/imports the project into STM32CubeIDE.

After compiling and loading the application, the user can observe the output of the application operation in the serial terminal. The application is described in the following sections.

2.3.1 NDEF_RW

The NDEF_RW application uses the NDEF middleware on top of RFAL to implement NDEF read/write/format procedures.

To execute correctly this demonstration the user needs some NFC forum formatted tags such as T2T, T3T, T4AT, T4BT or T5T. The user can choose for example tags of the ST25TA and ST25TV family, which can also be formatted using this application as explained in the below procedures.

The application waits for an NFC tag to be detected. By default, it reads its content. The user can press the blue user button to cycle among the different features:

- Write a text record
- Write a URI record and an Android™ application record (AAR)
- Format an ST tag
- NDEF records are read and decoded, and their content is displayed and tuned to their type, as stored in a message and written to the tag.

2.3.2 Card emulation and Bluetooth® pairing application

This application uses the NDEF middleware to demonstrate Bluetooth® connection handover. It creates an NFC Forum handover request message, and uses the RFAL to implement card emulation thanks to static pairing. This pairing record can be read for example by an NFC Android™ phone that initiates a Bluetooth® pairing procedure. It supports both T3T and T4T.

3 NDEF middleware

The NDEF library provides an API to handle NDEF message and records.

The software design is split in RF technology-independent and RF technology-dependent layers:

- The RF technology independent layer provides message, record and supported types management API
- The RF technology-dependent layer is made of:
  - An NDEF wrapper to provide a common API to the underlying NDEF T2T/T3T/T4A/T4BT/T5T technologies
  - The NDEF layer supporting T2T/T3T/T4A/T4BT/T5T technologies, based on the RFAL

The library provides an easy access API for the most common types:

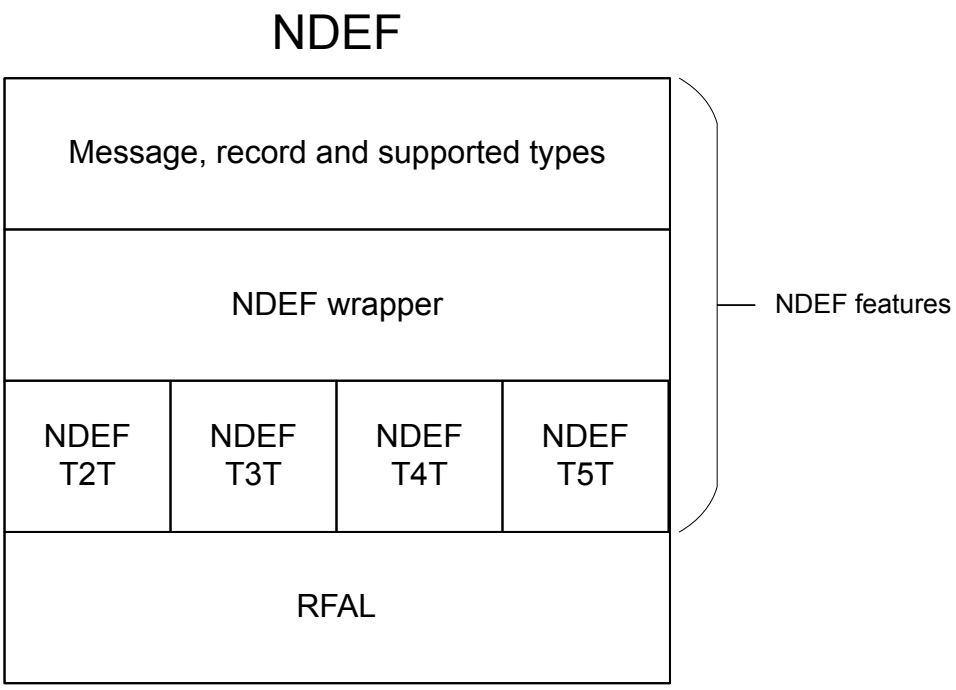
- RTD device information
- RTD text
- RTD URI
- RTD WLC (wireless charging capability, status information, poll information and listen control)
- AAR (Android™ application record)
- vCard (visit card)
- Bluetooth® BR/EDR and LE (basic rate/enhanced data rate, low energy and secure variants)
- Wi-Fi®
- Empty and flat types

See Table 2 for more details about supported types.

The main features of the library are:

- Scalability through compile time switches:
  - Enable read or read/write APIs
  - Embed only the NDEF types required
  - Ability to remove well-known type support for applications that do not need NDEF type handling
- No adherence between the NDEF wrapper and the message layers (i.e. the message/record/type API can easily be reused)

Figure 1. **NDEF block diagram**



The NDEF library is built on top of RFAL.

3.1 NDEF APIs

Detailed technical information about the NDEF APIs is available to the user and can be found in a compiled CHM files, located inside the ndef\doc folders of the software package where all the functions and parameters are fully described.

3.1.1 Library configuration

By default, all the supported types are enabled, and controlled by ndef_config.h.

To reduce the memory footprint, the user can define NDEF_CONFIG_CUSTOM and create the file ndef_config_custom.h to turn on only the desired type(s).

3.1.2 Layer

NDEF poller layers

The following tables describe the NDEF poller APIs.

Table 3. NDEF poller context initialization

Function	ReturnCode ndefPollerContextInitialization(ndefContext *ctx, const ndefDevice *dev);
Parameters	ctx: NDEF context dev: NDEF device
Returns	ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	This method performs the initialization of the NDEF context. It must be called after a successful anti-collision procedure and prior to any NDEF procedures such as NDEF detection procedure.

Table 4. NDEF poller NDEF detect

Function	ReturnCode ndefPollerNdefDetect(ndefContext *ctx, ndefInfo *info);
Parameters	ctx: NDEF context info: NDEF information (optional parameter, NULL may be used when no NDEF information is needed)
Returns	ERR_WRONG_STATE: library not initialized or mode not set ERR_REQUEST: detection failed ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	This method performs the NDEF detection procedure

Table 5. NDEF poller read bytes

Function	ReturnCode ndefPollerReadBytes(ndefContext *ctx, uint32_t offset, uint32_t len, uint8_t *buf, uint32_t *rcvdLen);
Parameters	ctx: NDEF context offset: file offset of where to start reading data len: requested length buf: buffer to place the data read from the tag rcvdLen: received length
Returns	ERR_WRONG_STATE: library not initialized or mode not set ERR_REQUEST: read failed ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	This method reads arbitrary length data

Table 6. NDEF poller write bytes

Function	ReturnCode ndefPollerWriteBytes(ndefContext *ctx, uint32_t offset, const uint8_t *buf, uint32_t len);
Parameters	ctx: NDEF context offset: file offset of where to start writing data buf: data to write len: buf length
Returns	ERR_WRONG_STATE: library not initialized or mode not set ERR_REQUEST: read failed ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	This method writes arbitrary length data from the current selected file

Table 7. NDEF poller read raw message

Function	ReturnCode ndefPollerReadRawMessage(ndefContext *ctx, uint8_t *buf, uint32_t bufLen, uint32_t *rcvdLen);
Parameters	ctx: NDEF context buf: buffer to place the NDEF message bufLen: buffer length rcvdLen: received length
Returns	ERR_WRONG_STATE: library not initialized or mode not set ERR_REQUEST: read failed ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	This method reads a raw NDEF message. Prior to NDEF read procedure, a successful ndefPollerNdefDetect() has to be performed.

Table 8. NDEF poller write raw message

Function	ReturnCode ndefPollerWriteRawMessage(ndefContext *ctx, const uint8_t *buf, uint32_t bufLen);
Parameters	ctx: NDEF context buf: raw message buffer bufLen: buffer length
Returns	ERR_WRONG_STATE: library not initialized or mode not set ERR_REQUEST:write failed ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	This method writes a raw NDEF message. Prior to NDEF write procedure, a successful ndefPollerNdefDetect() has to be performed.

Table 9. NDEF poller tag format

Function	ReturnCode ndefPollerTagFormat(ndefContext *ctx, const ndefCapabilityContainer *cc, uint32_t options);
Parameters	ctx: NDEF context cc: capability container options: specific flags
Returns	ERR_WRONG_STATE: library not initialized or mode not set ERR_REQUEST: write failed ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	This method format a tag to make it ready for NDEF storage. cc and options parameters usage is described in each technology method (ndefT[2345]TPollerTagFormat)

Table 10. NDEF poller write raw message len

Function	ReturnCode ndefPollerWriteRawMessageLen(ndefContext *ctx, uint32_t rawMessageLen);
Parameters	ctx: NDEF context rawMessageLen: len
Returns	ERR_WRONG_STATE: library not initialized or mode not set ERR_REQUEST: write failed ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	This method writes the NLEN field

Table 11. NDEF poller write message

Function	ReturnCode ndefPollerWriteMessage(ndefContext *ctx, const ndefMessage *message);
Parameters	ctx: NDEF context message: message to write
Returns	ERR_WRONG_STATE: library not initialized or mode not set ERR_REQUEST: write failed ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	Write the NDEF message to the tag

Table 12. NDEF poller check presence

Function	ReturnCode ndefPollerCheckPresence(ndefContext *ctx);
Parameters	ctx: NDEF context
Returns	ERR_WRONG_STATE: library not initialized or mode not set ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	This method checks whether an NFC tag is still present in the operating field

Table 13. NDEF poller check available space

Function	ReturnCode ndefPollerCheckPresence(ndefContext *ctx);
Parameters	ctx: NDEF context
Returns	ERR_WRONG_STATE: library not initialized or mode not set ERR_PARAM: invalid parameter ERR_PROTO: protocol error ERR_NONE: no error
Description	This method checks whether an NFC tag is still present in the operating field

Table 14. NDEF poller begin write message

Function	ReturnCode ndefPollerBeginWriteMessage(ndefContext *ctx, uint32_t messageLen);
Parameters	ctx: NDEF context messageLen: message length
Returns	ERR_PARAM: invalid parameter ERR_NOMEM: not enough space ERR_NONE: enough space for message of messageLen length
Description	This method sets the L-field to 0 (T1T, T2T, T4T, T5T) or set the WriteFlag (T3T) and sets the message offset to the proper value according to messageLen

Table 15. NDEF poller end write message

Function	ReturnCode ndefPollerEndWriteMessage(ndefContext *ctx, uint32_t messageLen);
Parameters	ctx: NDEF context messageLen: message length
Returns	ERR_PARAM: invalid parameter ERR_NOMEM: not enough space ERR_NONE: enough space for message of messageLen length
Description	This method updates the L-field value after the message has been written and resets the write flag (for T3T only)

Table 16. **NDEF poller set read only**

Function	ReturnCode ndefPollerSetReadOnly(ndefContext *ctx);
Parameters	ctx: NDEF context
Returns	ERR_PARAM: invalid parameter ERR_NONE: enough space for message of messageLen length
Description	This method performs the transition from the READ/WRITE state to the READ-ONLY state

Message layer

The following tables describe the NDEF message APIs.

Table 17. **NDEF message init**

Function	ReturnCode ndefMessageInit(ndefMessage* message);
Parameters	Message to inizialize
Returns	ERR_NONE if successful or a standard error code
Description	Initialize an empty NDEF message

Table 18. **NDEF message get info**

Function	ReturnCode ndefMessageGetInfo(const ndefMessage* message, ndefMessageInfo* info);
Parameters	Message to get info from info: e.g. message length in bytes, number of records
Returns	ERR_NONE if successful or a standard error code
Description	Get NDEF message information

Table 19. **NDEF message get record count**

Function	uint32_t ndefMessageGetRecordCount(const ndefMessage* message);
Parameters	Message
Returns	The number of records in the given message
Description	Get the number of NDEF message records

Table 20. **NDEF message append**

Function	ReturnCode ndefMessageAppend(ndefMessage* message, ndefRecord* record);
Parameters	record: Record to append message: message to be appended with the given record
Returns	ERR_NONE if successful or a standard error code
Description	Append a record to an NDEF message

Table 21. **NDEF message decode**

Function	ReturnCode ndefMessageDecode(const ndefConstBuffer* bufPayload, ndefMessage* message);
Parameters	bufPayload: payload buffer to convert into message message: message created from the raw buffer
Returns	ERR_NONE if successful or a standard error code
Description	Decode a raw buffer to an NDEF message

Table 22. **NDEF message encode**

Function	ReturnCode ndefMessageEncode(const ndefMessage* message, ndefBuffer* bufPayload);
Parameters	message: message to convert bufPayload: output buffer to store the converted message. The input length provides the output buffer allocated length, used for parameter check to avoid overflow. In case the buffer provided is too short, it is updated with the required buffer length. On success, it is updated with the actual buffer length used to contain the converted message.
Returns	ERR_NONE if successful or a standard error code
Description	Encode an NDEF message to a raw buffer

Table 23. **NDEF message find record type**

Function	ndefRecord* ndefMessageFindRecordType(ndefMessage* message, uint8_t tnf, const ndefConstBuffer8* bufType);
Parameters	message: message to parse tnf: TNF type to match bufType: type buffer to match
Returns	The record matching the type if successful or NULL
Description	Parses an NDEF message, looking for a record of given type

Record layer

The following tables describe the record APIs.

Table 24. **NDEF record reset**

Function	ReturnCode ndefRecordReset(ndefRecord* record);
Parameters	record to reset
Returns	ERR_NONE if successful or a standard error code
Description	This function clears every record field

Table 25. NDEF record init

Function	ReturnCode ndefRecordInit(ndefRecord* record, uint8_t tnf, const ndefConstBuffer8* bufType, const ndefConstBuffer8* bufId, const ndefConstBuffer* bufPayload);
Parameters	record: record to initialize tnf: TNF type bufType: type buffer bufId: id buffer bufPayload: payload buffer
Returns	ERR_NONE if successful or a standard error code
Description	This function initializes all record fields

Table 26. NDEF record get header length

Function	uint32_t ndefRecordGetHeaderLength(const ndefRecord* record);
Parameters	record
Returns	Header length in bytes
Description	Return the length of header for the given record

Table 27. NDEF record get length

Function	uint32_t ndefRecordGetLength(const ndefRecord* record);
Parameters	record
Returns	Record length in bytes
Description	Return the length of the given record, needed to store it as a raw buffer. It includes the header length.

Table 28. NDEF record set type

Function	ReturnCode ndefRecordSetType(ndefRecord* record, uint8_t tnf, const ndefConstBuffer8* bufType);
Parameters	record: record to set the type tnf: TNF type bufType: type buffer
Returns	ERR_NONE if successful or a standard error code
Description	Set the type for the given record

Table 29. NDEF record get type

Function	bool ndefRecordTypeMatch(const ndefRecord* record, uint8_t tnf, const ndefConstBuffer8* bufType);
Parameters	record: record to get the type from tnf: the TNF type to compare with bufType: type string buffer to compare with
Returns	True or false
Description	Check the record type matches a given type

Table 30. NDEF record set id

Function	ReturnCode ndefRecordSetId(ndefRecord* record, const ndefConstBuffer8* bufId);
Parameters	record: record to set the id bufId: id buffer bufId: id buffer
Returns	ERR_NONE if successful or a standard error code
Description	Set the id for the given NDEF record

Table 31. NDEF record get id

Function	ReturnCode ndefRecordGetId(const ndefRecord* record, ndefConstBuffer8* bufId);
Parameters	record: Record to get the id from bufId: id buffer
Returns	ERR_NONE if successful or a standard error code
Description	Return the id for the given NDEF record

Table 32. NDEF record set payload

Function	ReturnCode ndefRecordSetPayload(ndefRecord* record, const ndefConstBuffer* bufPayload);
Parameters	record: record to set the payload bufPayload: payload buffer
Returns	ERR_NONE if successful or a standard error code
Description	Set the payload for the given record, update the SR bit accordingly

Table 33. NDEF record get payload

Function	ReturnCode ndefRecordGetPayload(const ndefRecord* record, ndefConstBuffer* bufPayload);
Parameters	record: record to get the payload from bufPayload: payload buffer
Returns	ERR_NONE if successful or a standard error code
Description	Return the payload for the given record

Table 34. NDEF record decode

Function	ReturnCode ndefRecordDecode(const ndefConstBuffer* bufPayload, ndefRecord* record);
Parameters	record: record created from the raw buffer bufPayload: payload buffer to convert into record
Returns	ERR_NONE if successful or a standard error code
Description	Decode a raw buffer to create an NDEF record

Table 35. NDEF record encode header

Function	ReturnCode ndefRecordEncodeHeader(const ndefRecord* record, ndefBuffer* bufHeader);
Parameters	record: record header to convert bufHeader: output buffer to store the converted record header. The input length provides the output buffer allocated length, used for parameter check to avoid overflow. In case the buffer provided is too short, it is updated with the required buffer length. On success, it is updated with the actual buffer length used to contain the converted record.
Returns	ERR_NONE if successful or a standard error code
Description	Convert a record header to a raw buffer. It consists of: • "header byte" (1 byte), type length (1 byte) • payload length (4 bytes), Id length (1 byte) Total 7 bytes.

Table 36. NDEF record encode

Function	ReturnCode ndefRecordEncode(const ndefRecord* record, ndefBuffer* bufRecord);
Parameters	record: record to convert bufRecord: output buffer to store the converted record The input length provides the output buffer allocated length, used for parameter check to avoid overflow. In case the buffer provided is too short, it is updated with the required buffer length. On success, it is updated with the actual buffer length used to contain the converted record.
Returns	ERR_NONE if successful or a standard error code
Description	Convert a record to a raw buffer

Table 37. NDEF record get payload length

Function	uint32_t ndefRecordGetPayloadLength(const ndefRecord* record);
Parameters	record
Returns	Payload length in bytes
Description	Get NDEF record payload length

Table 38. NDEF record get payload item

Function	const uint8_t* ndefRecordGetPayloadItem(const ndefRecord* record, ndefConstBuffer* bufPayloadItem, bool begin);
Parameters	record: record bufPayloadItem: the payload item returned begin: tell to return the first payload item or the next one
Returns	ERR_NONE if successful or a standard error code
Description	Call this function to get either the first payload item, or the next one. Returns the next payload item, call it until it returns NULL.

3.1.3 Examples

Following sections give simple examples showing how to use the NDEF library.

Reading an NDEF message

```
void ndefExampleRead( void )
{
    ReturnCode err;
    uint32_t   rawMessageLen;

    /*
     * RFAL Init
     */
    err = rfalNfcInitialize();
    if( err != ERR_NONE )
    {
        platformLog("rfalNfcInitialize return %d\r\n", err);
        return;
    }

    rfalNfcDeactivate( false );
    rfalNfcDiscover( &discParam );

    /*
     * Read loop
     */
    while (1)
    {
        rfalNfcWorker();
        if( rfalNfcIsDevActivated(rfalNfcGetState()) )
        {
            /*
             * Retrieve NFC device
             */
            rfalNfcGetActiveDevice(&nfcDevice);

            /*
             * Perform NDEF Context Initialization
             */
            err = ndefPollerContextInitialization(&ndefCtx, nfcDevice);
            if( err != ERR_NONE )
            {
                platformLog("NDEF NOT DETECTED (ndefPollerContextInitialization returns %d)
\r\n", err);
                return;
            }

            /*
             * Perform NDEF Detect procedure
             */
            err = ndefPollerNdefDetect(&ndefCtx, NULL);
            if( err != ERR_NONE )
            {
                platformLog("NDEF NOT DETECTED (ndefPollerNdefDetect returns %d)\r\n", err);
                return;
            }

            /*
             * Perform NDEF read procedure
             */
            err = ndefPollerReadRawMessage(&ndefCtx, rawMessageBuf, sizeof(rawMessageBuf),
&rawMessageLen);
```

```
if( err != ERR_NONE )
            {
                platformLog("NDEF message cannot be read (ndefPollerReadRawMessage returns
%d)\r\n", err);
                return;
            }

            platformLog("NDEF Read successful\r\n");

            /*
             * Parse message content
             */
            ndefExampleParseMessage(rawMessageBuf, rawMessageLen);

            return;
        }
    }
}
```

```
/*!
 *****************************************************************************
 * \brief ndefExampleParseMessage
 *
 *  This function parses the NDEF message
 *****************************************************************************
 */
void ndefExampleParseMessage(uint8_t* rawMsgBuf, uint32_t rawMsgLen)
{
    ReturnCode       err;
    ndefConstBuffer  bufRawMessage;
    ndefMessage      message;
    ndefRecord*      record;

    bufRawMessage.buffer = rawMsgBuf;
    bufRawMessage.length = rawMsgLen;

    err = ndefMessageDecode(&bufRawMessage, &message);
    if (err != ERR_NONE)
    {
        return;
    }

    record = ndefMessageGetFirstRecord(&message);

    while (record != NULL)
    {
        ndefExampleParseRecord(record);
        record = ndefMessageGetNextRecord(record);
    }
}
```

```
/*!
 *****************************************************************************
 * \brief ndefExampleParseRecord
 *
 *  This function parses a given record
 *****************************************************************************
 */
void ndefExampleParseRecord(ndefRecord* record)
{
    ReturnCode       err;
    ndefType         type;

    err = ndefRecordToType(record, &type);
    if (err != ERR_NONE)
    {
        return;
    }

    switch (type.id)
    {
        case NDEF_TYPE_ID_EMPTY:
             platformLog(" * Empty record\r\n");
             break;
        case NDEF_TYPE_ID_RTD_DEVICE_INFO:
            platformLog(" * Device info record\r\n");
            break;
        case NDEF_TYPE_ID_RTD_TEXT:
             platformLog(" * TEXT record: ");
             ndefExamplePrintString(type.data.text.bufSentence.buffer,
type.data.text.bufSentence.length);
             break;
        case NDEF_TYPE_ID_RTD_URI:
             platformLog(" * URI record: ");
             ndefExamplePrintString(type.data.uri.bufUriString.buffer,
type.data.uri.bufUriString.length);
             break;
        case NDEF_TYPE_ID_RTD_AAR:
             platformLog(" * AAR record: ");
             ndefExamplePrintString(type.data.aar.bufPayload.buffer,
type.data.aar.bufPayload.length);
             break;
        case NDEF_TYPE_ID_RTD_WLCCAP:  /* Fall through */
        case NDEF_TYPE_ID_RTD_WLCSTAI: /* Fall through */
        case NDEF_TYPE_ID_RTD_WLCINFO: /* Fall through */
        case NDEF_TYPE_ID_RTD_WLCCTL:
            platformLog(" * WLC record\r\n");
            break;
        case NDEF_TYPE_ID_BLUETOOTH_BREDR:        /* Fall through */
        case NDEF_TYPE_ID_BLUETOOTH_LE:           /* Fall through */
        case NDEF_TYPE_ID_BLUETOOTH_SECURE_BREDR: /* Fall through */
        case NDEF_TYPE_ID_BLUETOOTH_SECURE_LE:    /* Fall through */
            platformLog(" * Bluetooth record\r\n");
            break;
        case NDEF_TYPE_ID_MEDIA_VCARD:
            platformLog(" * vCard record\r\n");
            break;
        case NDEF_TYPE_ID_MEDIA_WIFI:
            platformLog(" * WIFI record\r\n");
            break;
        default:
             platformLog(" * Other record\r\n");
             break;
     }
}
```

Creating an NDEF record

```
/*!
 *****************************************************************************
 * \brief ndefExampleWrite
 *
 *  This function performs the various NFC activities up to the NDEF tag writing
 *****************************************************************************
 */
void ndefExampleWrite( void )
{
    ReturnCode      err;
    ndefConstBuffer bufUri;
    ndefType        uri;
    ndefRecord      record;
    ndefMessage     message;
    static uint8_t  ndefURI[] = "st.com";

    /*
     * Message creation
     */
    err  = ndefMessageInit(&message);  /* Initialize message structure */
    bufUri.buffer = ndefURI;
    bufUri.length = strlen((char *)ndefURI);
    err |= ndefRtdUriInit(&uri, NDEF_URI_PREFIX_HTTP_WWW, &bufUri); /* Initialize URI type
structure */
    err |= ndefRtdUriToRecord(&uri, &record); /* Encode URI Record */
    err |= ndefMessageAppend(&message, &record); /* Append URI to message */
    if( err != ERR_NONE )
    {
        platformLog("Message creation failed\r\n", err);
        return;
    }

    /*
     * RFAL Init
     */
    err = rfalNfcInitialize();
    if( err != ERR_NONE )
    {
        platformLog("rfalNfcInitialize return %d\r\n", err);
        return;
    }

    rfalNfcDeactivate( false );
    rfalNfcDiscover( &discParam );

    /*
     * Write loop
     */
    while (1)
    {
        rfalNfcWorker();
        if( rfalNfcIsDevActivated(rfalNfcGetState()) )
        {
            /*
             * Retrieve NFC device
             */
            rfalNfcGetActiveDevice(&nfcDevice);

            /*
             * Perform NDEF Context Initialization
             */
            err = ndefPollerContextInitialization(&ndefCtx, nfcDevice);
            if( err != ERR_NONE )
            {
                platformLog("NDEF NOT DETECTED (ndefPollerContextInitialization returns %d)
\r\n", err);
```

```
                return;
            }

            /*
             * Perform NDEF Detect procedure
             */
            err = ndefPollerNdefDetect(&ndefCtx, NULL);
            if( err != ERR_NONE )
            {
                platformLog("NDEF NOT DETECTED (ndefPollerNdefDetect returns %d)\r\n", err);
                return;
            }

            /*
             * Perform NDEF write procedure
             */
            err = ndefPollerWriteMessage(&ndefCtx, &message);
            if( err != ERR_NONE )
            {
                platformLog("NDEF message cannot be written (ndefPollerReadRawMessage
returns %d)\r\n", err);
                return;
            }

            platformLog("NDEF Write successful\r\n");
            return;
        }
    }
}
```

Revision history

Table 39. **Document revision history**

Date	Version	Changes
21-Apr-2021	1	Initial release.

Contents

List of tables

List of figures