TEXAS INSTRUMENTS

TivaWare™ nfclib

USER'S GUIDE

SW-TM4C-NFCLIB-UG-2.1.2.111

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
www.ti.com/tiva-c

Revision Information

This is version 2.1.2.111 of this document, last updated on December 16, 2015.

Table of Contents

1 Introduction

This document describes the functions and variables of the NFC API and how they are used in the TivaWare environment. The NFC API assumes that the TRF7970A transciever is being used for connecting to NFC cards. Because signal connections vary among different boards and booster-packs, the NFC API requires a hardware mapping file with pin and port definitions. An example of this file is provided in nfclib/trf79x0.hw. Each application that uses nfclib should provide its own mapping file.

2 Using nfclib

This section describes the basics of using the NFC APIs to communicate with an NFC P2P Device using the TRF7970ATB module. Before any other NFC APIs can be called, the application must make a call to TRF79x0Init(), which configures the pins and the SSI port as defined in the application-supplied trf79x0_hw.h file. Once TRF79x0Init() is called by the application, the application attempts to initialize the NFC P2P state machine mode and frequency with a call to the NFCP2P_init() function. In addition to these functions, it is necessary to implement TimerSet(), TimerDisable(), and TimerInteruptHandler() functions in the user code. The NFCP2P_proccessStateMachine() function can and should be called periodically to determine if a new device has entered the field and is accessible. The NFCP2P_getReceiveState() function returns a structure with the current receive state and a pointer to the raw data buffer. This raw data buffer is volatile and changes each time a new NFC device comes into the field. The pointer to the raw data buffer should be given to NFCP2P_NDEFMessageDecoder(). This function returns a structure that contains the header information about the message as well as a pointer to the payload buffer. Once NFCP2P_NDEFMessageDecoder() has been called, the raw data is pulled into a non-volatile buffer so it can be processed by the record decoders. One part of the header information is the record type, for which a record decoder should be called to interpret the payload data. The length of the payload is another field in the header data. NFC data can be decoded in this manner. To encode NFC data, first use the record encoders to encode a record structure to a buffer. Then set the payload pointer in a sNDEFMessageData structure to the buffer as well as the other message header fields. Then call NFCP2P_NDEFMessageEncoder() to encode all the data into another buffer. This final buffer with all the NFC data encoded into a raw NFC format can then be sent using the NFCP2P_sendPacket() function.

Example: Initialize NFC P2P and process loop

```
//
// Variable to hold current mode the NFC stack is in, initialize to passive.
//
tTRF79x0TRFMode     eCurrentTRF79x0Mode = P2P_PASSIVE_TARGET_MODE;

//
// Receive status object from low level SNEP/NFC stack
//
sNFCP2PRxStatus     TRFReceiveStatus;

//
// Initialize the TRF79x0 and SSI
//
TRF79x0Init();

//
// Initialize Timer0A
//
Timer0AInit();

//
// Enable First Mode
//
NFCP2P_init(eCurrentTRF79x0Mode,FREQ_212_KBPS);

//
// Enable interrupts
//
IntMasterEnable();

//
// Call NFCP2P_proccessStateMachine() periodically and process data accordingly
```

```
//
while(1)
{
    //
    // Flip between Initiator and Passive mode to catch different tag types
    // in field
    //
    if(NFCP2P_proccessStateMachine() == NFC_P2P_PROTOCOL_ACTIVATION)
    {
        if(eCurrentTRF79x0Mode == P2P_INITATIOR_MODE)
        {
            eCurrentTRF79x0Mode = P2P_PASSIVE_TARGET_MODE;
            //Toggle LED
        }
        else if(eCurrentTRF79x0Mode == P2P_PASSIVE_TARGET_MODE)
        {
            eCurrentTRF79x0Mode = P2P_INITATIOR_MODE;
            //Toggle LED
        }

        //
        // Initiator switch to target mode or vice versa
        //
        NFCP2P_init(eCurrentTRF79x0Mode,FREQ_212_KBPS);
    }

    //
    // Read the receive status structure - check if there is a received
    // packet from the target
    //
    TRFReceiveStatus = NFCP2P_getReceiveState();

    //
    // Process short records
    //
    if(TRFReceiveStatus.eDataReceivedStatus != RECEIVED_NO_FRAGMENT)
    {
        //
        // Message Decoding / Encoding / Handling goes here
        //
    }
    //
    // Process fragmented / chunked records
    //
    else if(TRFReceiveStatus.eDataReceivedStatus != RECEIVED_FIRST_FRAGMENT)
    {
        //
        // Message decoding / encoding / handling goes here
        //
    }
    else if(TRFReceiveStatus.eDataReceivedStatus != RECEIVED_N_FRAGMENT)
    {
        //
        // Message decoding / encoding / handling goes here
        //
    }
}
```

Example: Decode NFC P2P Message

This example demonstrates how to decode the raw buffer into message and record structures that can be processed. The header information should be decoded first because it contains the length and type of the payload record.

Note:

It is assumed that by this point NFCP2P_getReceiveState() has been called and has the data to be processed in a sNFCP2PRxStatus structure. It is also assumed that the message being decoded is a short record and is completely encoded in a single buffer.

```
//
// NFC NDEF message containers. These structures are used in combination with
// the decode functions to extract data out of a raw NFC data buffer. They are
// also used with the encode functions to recreate the raw data in preperation
// for sending it.
//
sNDEFMessageData        sNDEFMessage;
sNDEFTextRecord         sNDEFText;
sNDEFURIRecord          sNDEFURI;
sNDEFSmartPosterRecord  sNDEFSmartPoster;

//
// Receive status object from low level SNEP/NFC stack
//
sNFCP2PRxStatus         TRFReceiveStatus;

//
// Index / counter variable
//
uint32_t ui32x;
uint32_t ui32Type;

//
// Decode message header from buffer to structure
//
NFCP2P_NDEFMessageDecoder(&sNDEFMessage,  TRFReceiveStatus.pui8RxDataPtr);

//
// Extract the TypeID into a single number
//
for(ui32x=0,ui32Type=0;ui32x<sNDEFMessage.ui8TypeLength;ui32x++)
{
    ui32Type=(ui32Type<<8)+sNDEFMessage.pui8Type[ui32x];
}

//
// Handler for different message types
//
switch(ui32Type)
{
    //
    // Text Record 'T'
    //
    case NDEF_TYPE_TEXT:
        //
        // Decode the Record from buffer to structure
        //
        NFCP2P_NDEFTextRecordDecoder(&sNDEFText,    \
                                    sNDEFMessage.pui8PayloadPtr, \
                                    sNDEFMessage.ui32PayloadLength);
        break;

    //
    // URI Record 'U'
    //
    case NDEF_TYPE_URI:
        //
        // Decode the record from buffer to structure
        //
        NFCP2P_NDEFURIRecordDecoder(&sNDEFURI,  \
                                    sNDEFMessage.pui8PayloadPtr,  \
                                    sNDEFMessage.ui32PayloadLength);
        break;

    //
    // SmartPoster Record 'Sp'
    //
    case NDEF_TYPE_SMARTPOSTER:
        //
        // Decode the record from buffer to structure
        //
        NFCP2P_NDEFSmartPosterRecordDecoder(&sNDEFSmartPoster, \
                                    sNDEFMessage.pui8PayloadPtr, \
                                    sNDEFMessage.ui32PayloadLength);
        break;

    default:
        //
        // Tag type not recognized. Add error handler here.
        //
        break;
}
```

Example: Encode NFC P2P Message

This code snippet shows the order of operations to encode an NDEFMessage and NDEFRecord structures into a raw buffer to send over NFC. It is assumed that the NDEFMessage structure and NDEFRecord structure have been filled out appropriately and linked together before this code is called. First the Record is encoded and then the Message Header is encoded as a wrapper around it.

```
//
// Simple Variables
//
uint32_t ui32x;              // used as index
uint32_t ui32Type            // numeric conversion of NDEFRecord Type
uint32_t ui32RecordLength    // length of payload encoded into the Record Buffer
uint32_t ui32MessageLength   // length of message encoded in Message Buffer
                             // (Record + Header)

//
// NFC NDEF message containers. These structures are used in combination with
// the decode functions to extract data out of a raw NFC data buffer. They are
// also used with the encode functions to recreate the raw data in preparation
// for sending it.
//
sNDEFMessageData        sNDEFMessage;
sNDEFTextRecord         sNDEFText;
sNDEFURIRecord          sNDEFURI;
sNDEFSmartPosterRecord  sNDEFSmartPoster;

//
// Buffers to hold raw data, the lengths are arbitrary and can be of any
// sufficiently long length.
//
uint8_t pui8MessageBuffer[100];
uint8_t pui8RecordBuffer[100];

//
// Determine tag type from NDEFMessage 'pui8Type' field
//
for(ui32x=0,Type=0;ui32x<sNDEFMessage.ui8TypeLength;ui32x++)
{
    Type=(Type<<8)+sNDEFMessage.pui8Type[ui32x];
}

//
// Encode different supported record types to the recordbuffer
//
switch(Type)
{
    case NDEF_TYPE_TEXT :
        //
        // Encode the record from structure to buffer
        //
        NFCP2P_NDEFTextRecordEncoder(sNDEFText, RecordBuffer,  \
                                        &RecordLength);
        break;
    case NDEF_TYPE_URI :
        //
        // Encode the record from structure to buffer
        //
        NFCP2P_NDEFURIRecordEncoder(sNDEFURI, RecordBuffer, \
                                        &RecordLength);
        break;
    case NDEF_TYPE_SMARTPOSTER :
        //
        // Encode the record from structure to buffer
        //
        NFCP2P_NDEFSmartPosterRecordEncoder(sNDEFSmartPoster, \
                                            RecordBuffer,     \
                                            &RecordLength);
        break;
    default:
        //
        // The tag type is unrecognized. Do error handling here
        // The code should not continue past this point as there is no
        // function to encode the record.
        //
        break;

//
// Point payload pointer to encoded payload buffer
//
sNDEFMessage.pui8PayloadPtr=RecordBuffer;

//
// Set length of payload
//
sNDEFMessage.ui32PayloadLength=RecordLength;

//
// Encode message header from structure to buffer, store length in MessageLength
//  This is used to echo the tag back over NFC
//
NFCP2P_NDEFMessageEncoder(sNDEFMessage, MessageBuffer,&MessageLength);

//
// Send the NFC data to the NFC stack for processing.
//
NFCP2P_sendPacket(MessageBuffer,MessageLength);
```

Example: Encode NFC P2P URI Record

This example demonstrates how to fill out a URI Record structure so it can be encoded.

```
    //
    // NFC NDEF message containers. These structures are used in combination with the
```

```
// encode functions to recreate the raw data in preperation for sending it.
//
sNDEFMessageData        NDEFMessage;
sNDEFURIRecord          NDEFURI;

//
// URI string to send and variable to hold length of payload.
//
uint8_t  ui8TIWebpage[]="ti.com/tm4c129xkit";
uint32_t ui32RecordLength=0, MessageLength=0;

//
// Set Header Information
// This message is one record long, sent in one burst, is short,
// has no ID field, and is a well known type of record.
// For more information on how to use these fields, please see the
// NFC Data Exchange Format (NDEF) Technical Specification
// at http://www.nfc-forum.org/specs/spec_list/
//
sNDEFMessage.sStatusByte.MB=1;     // Message Begin
sNDEFMessage.sStatusByte.ME=1;     // Message End
sNDEFMessage.sStatusByte.CF=0;     // Record is Not Chunked
sNDEFMessage.sStatusByte.SR=1;     // Record is Short Record
sNDEFMessage.sStatusByte.IL=0;     // ID Length =0 (No ID Field Present)
sNDEFMessage.sStatusByte.TNF=TNF_WELLKNOWNTYPE;

//
// Set Type to URI ('U')
// Set Type lengh to one character
//
sNDEFMessage.pui8Type[0]='U';              // 'U' is the Type for URI's
sNDEFMessage.ui8TypeLength=1;              // TypeLengh is 1 char long ('U')
//sNDEFMessage.ui8IDLength= ;              //not needed, IL=0 so no ID is given
//sNDEFMessage.pui8ID=;                    //not needed, IL=0 so no ID is given

//
// Set URI Record Info
//
// Prepend the URI with 'http://www.' (see nfc_p2p.h for a full list of options)
// Set the string pointer to the webpage string
// Set the length of the string
//
sNDEFURI.eIDCode=http_www;
sNDEFURI.puiUTF8String=ui8TIWebpage;
sNDEFURI.ui32URILength=sizeof(ui8TIWebpage);

//
// Encode the URI record into the payload buffer, returns the length
// of the buffer written in the ui32RecordLength variable
//
NFCP2P_NDEFURIRecordEncoder(sNDEFURI,pui8RecordBuffer,&ui32RecordLength);

//
// Set the length of the payload and the payload pointer in the
// header structure
//
sNDEFMessage.ui32PayloadLength=ui32RecordLength;
sNDEFMessage.pui8PayloadPtr=pui8RecordBuffer;

//
// Encode the header and the payload into the message buffer
//
NFCP2P_NDEFMessageEncoder(sNDEFMessage,pui8MessageBuffer,&MessageLength);

//
// Send the NFC message data to the stack for processing
```

```
//
NFCP2P_sendPacket(pui8MessageBuffer,MessageLength);
```

3 Hardware Customization

This section covers the customizable settings that can be changed in the trf79x0_hw.h file. These definitions are controlled through compile-time defines and allow changing the SSI and GPIO control signals. The trf79x0_hw_example.h file is included as an example and it assumes a TRF7970ATB EM board is connected to a DK-TM4C129X development kit on the EM header.

Note:
The default defines are for the TFR7970ATB EM adapter that is used with the DK-TM4C129X Development Platform and may not match your platform.

3.1 NFC Hardware Definitions

Defines

- SSI_CLK_RATE
- TRF79X0_ASKOK_BASE
- TRF79X0_ASKOK_PERIPH
- TRF79X0_ASKOK_PIN
- TRF79X0_CLK_BASE
- TRF79X0_CLK_CONFIG
- TRF79X0_CLK_PERIPH
- TRF79X0_CLK_PIN
- TRF79X0_CS_BASE
- TRF79X0_CS_PERIPH
- TRF79X0_CS_PIN
- TRF79X0_EN2_BASE
- TRF79X0_EN2_PERIPH
- TRF79X0_EN2_PIN
- TRF79X0_EN_BASE
- TRF79X0_EN_PERIPH
- TRF79X0_EN_PIN
- TRF79X0_IRQ_BASE
- TRF79X0_IRQ_INT
- TRF79X0_IRQ_PERIPH
- TRF79X0_IRQ_PIN
- TRF79X0_MOD_BASE
- TRF79X0_MOD_PERIPH
- TRF79X0_MOD_PIN
- TRF79X0_RX_BASE
- TRF79X0_RX_CONFIG
- TRF79X0_RX_PERIPH
- TRF79X0_RX_PIN
- TRF79X0_SSI_BASE
- TRF79X0_SSI_PERIPH

- TRF79X0_TX_BASE
- TRF79X0_TX_CONFIG
- TRF79X0_TX_PERIPH
- TRF79X0_TX_PIN

3.1.1 Detailed Description

This section covers the definitions that control which hardware is used to communicate with the TRF79x0 EM module. These defines configure which SSI peripheral is used as well as which pins are assigned to the other connections to the TRF79x0 EM module. The **TRF79X0_SSI_**∗ defines are used to specify the SSI peripheral that is used by the application. The remaining defines specify the pins used by the NFC APIs. The TRF79x0 EM module requires the following signal connections: CLK, RX, TX, CS, ASKOK, EN, EN2, IRQ, MOD. To configure these signals, three defines must be set for each. For example, for the CS signal, the TRF79X0_CS_BASE, TRF79X0_CS_PERIPH and TRF79X0_CS_PIN defines must be set.

Example: CS pin is on GPIO port E pin 1.

```
#define TRF79X0_CS_BASE         GPIO_PORTA_BASE
#define TRF79X0_CS_PERIPH       SYSCTL_PERIPH_GPIOA
#define TRF79X0_CS_PIN          GPIO_PIN_4
```

3.1.2 Define Documentation

3.1.2.1 SSI_CLK_RATE

Definition:

```
#define SSI_CLK_RATE
```

Description:

The clock rate of the SSI clock specified in Hz.

Example: 2-MHz SSI data clock.

```
#define SSI_CLK_RATE 2000000
```

3.1.2.2 TRF79X0_ASKOK_BASE

Definition:

```
#define TRF79X0_ASKOK_BASE
```

Description:

Specifies the *base* address of the GPIO port that is connected to the ASKOK signal on the TRF79x0 EM board.

Example: The ASKOK signal is on GPIO port J.

```
#define TRF79X0_ASKOK_BASE GPIO_PORTJ_BASE
```

3.1.2.3 TRF79X0_ASKOK_PERIPH

Definition:

```
#define TRF79X0_ASKOK_PERIPH
```

Description:

Specifies the *peripheral* for the GPIO port that is connected to the ASKOK signal on the TRF79x0 EM board.

Example: The ASKOK signal is on GPIO port J.

```
#define TRF79X0_ASKOK_PERIPH SYSCTL_PERIPH_GPIOJ
```

3.1.2.4 TRF79X0_ASKOK_PIN

Definition:

```
#define TRF79X0_ASKOK_PIN
```

Description:

Specifies the GPIO pin that is connected to the ASKOK signal on the TRF79x0 EM board.

Example: The ASKOK signal is on GPIO pin 5.

```
#define TRF79X0_ASKOK_PIN GPIO_PIN_5
```

3.1.2.5 TRF79X0_CLK_BASE

Definition:

```
#define TRF79X0_CLK_BASE
```

Description:

Specifies the *base* address of the GPIO port that is connected to the SSI Clock signal on the TRF79x0 EM board.

Example: The SSI peripheral CLK signal is on GPIO port A.

```
#define TRF79X0_CLK_BASE GPIO_PORTA_BASE
```

3.1.2.6 TRF79X0_CLK_CONFIG

Definition:

```
#define TRF79X0_CLK_CONFIG
```

Description:

Specifies the GPIO pin that is connected to the SSI Clock signal on the TRF79x0 EM board.

Example: The SSI Clock signal is on GPIO port A pin 2.

```
#define TRF79X0_CLK_CONFIG GPIO_PA2_SSI0CLK
```

3.1.2.7 TRF79X0_CLK_PERIPH

Definition:

```
#define TRF79X0_CLK_PERIPH
```

Description:

Specifies the *peripheral* for the GPIO port that is connected to the SSI Clock signal on the TRF79x0 EM board.

Example: The SSI peripheral CLK signal is on GPIO port A.

```
#define TRF79X0_CLK_PERIPH SYSCTL_PERIPH_GPIOA
```

3.1.2.8 TRF79X0_CLK_PIN

Definition:

```
#define TRF79X0_CLK_PIN
```

Description:

Specifies the GPIO pin that is connected to the SSI Clock signal on the TRF79x0 EM board.

Example: The SSI peripheral CLK signal is on GPIO pin 2.

```
#define TRF79X0_CLK_PIN GPIO_PIN_2
```

3.1.2.9 TRF79X0_CS_BASE

Definition:

```
#define TRF79X0_CS_BASE
```

Description:

Specifies the *base* address of the GPIO port that is connected to the SSI CS signal on the TRF79x0 EM board.

Example: The SSI CS signal is on GPIO port A.

```
#define TRF79X0_CS_BASE GPIO_PORTA_BASE
```

3.1.2.10 TRF79X0_CS_PERIPH

Definition:

```
#define TRF79X0_CS_PERIPH
```

Description:

Specifies the *peripheral* for the GPIO port that is connected to the SSI CS signal on the TRF79x0 EM board.

Example: The SSI CS signal is on GPIO port A.

```
#define TRF79X0_CS_PERIPH SYSCTL_PERIPH_GPIOA
```

3.1.2.11 TRF79X0_CS_PIN

Definition:

```
#define TRF79X0_CS_PIN
```

Description:

Specifies the GPIO pin that is connected to the SSI CS signal on the TRF79x0 EM board.

Example: The SSI peripheral CS signal is on GPIO pin 4.

```
#define TRF79X0_CS_PIN GPIO_PIN_4
```

3.1.2.12 TRF79X0_EN2_BASE

Definition:

```
#define TRF79X0_EN2_BASE
```

Description:

Specifies the *base* address of the GPIO port that is connected to the EN2 signal on the TRF79x0 EM board.

Example: The EN2 signal is on GPIO port D.

```
#define TRF79X0_EN2_BASE GPIO_PORTD_BASE
```

3.1.2.13 TRF79X0_EN2_PERIPH

Definition:

```
#define TRF79X0_EN2_PERIPH
```

Description:

Specifies the *peripheral* for the GPIO port that is connected to the EN2 signal on the TRF79x0 EM board.

Example: The EN2 signal is on GPIO port D.

```
#define TRF79X0_EN2_PERIPH SYSCTL_PERIPH_GPIOD
```

3.1.2.14 TRF79X0_EN2_PIN

Definition:

```
#define TRF79X0_EN2_PIN
```

Description:

Specifies the GPIO pin that is connected to the EN2 signal on the TRF79x0 EM board.

Example: The EN2 signal is on GPIO pin 3.

```
#define TRF79X0_EN2_PIN GPIO_PIN_3
```

3.1.2.15 TRF79X0_EN_BASE

Definition:
```
#define TRF79X0_EN_BASE
```

Description:
Specifies the *base* address of the GPIO port that is connected to the EN signal on the TRF79x0 EM board.

Example: The EN signal is on GPIO port D.

```
#define TRF79X0_EN_BASE GPIO_PORTD_BASE
```

3.1.2.16 TRF79X0_EN_PERIPH

Definition:
```
#define TRF79X0_EN_PERIPH
```

Description:
Specifies the *peripheral* for the GPIO port that is connected to the EN signal on the TRF79x0 EM board.

Example: The EN signal is on GPIO port D.

```
#define TRF79X0_EN_PERIPH SYSCTL_PERIPH_GPIOD
```

3.1.2.17 TRF79X0_EN_PIN

Definition:
```
#define TRF79X0_EN_PIN
```

Description:
Specifies the GPIO pin that is connected to the EN pin on the TRF79x0 EM board.

Example: The EN signal is on GPIO pin 2.

```
#define TRF79X0_EN_PIN GPIO_PIN_2
```

3.1.2.18 TRF79X0_IRQ_BASE

Definition:
```
#define TRF79X0_IRQ_BASE
```

Description:
Specifies the *base* address of the GPIO port that is connected to the IRQ signal on the TRF79x0 EM board.

Example: The IRQ signal is on GPIO port J.

```
#define TRF79X0_IRQ_BASE GPIO_PORTJ_BASE
```

3.1.2.19 TRF79X0_IRQ_INT

Definition:
```
#define TRF79X0_IRQ_INT
```

Description:
Specifies GPIO interrupt that is tied to the GPIO port that the IRQ signal is connected to TRF79x0 EM board.

Example: SSI GPIO interrupt is on GPIO port C.

```
#define TRF79X0_IRQ_INT INT_GPIOC
```

3.1.2.20 TRF79X0_IRQ_PERIPH

Definition:
```
#define TRF79X0_IRQ_PERIPH
```

Description:
Specifies the *peripheral* for the GPIO port that is connected to the IRQ signal on the TRF79x0 EM board.

Example: The IRQ signal is on GPIO port J.

```
#define TRF79X0_IRQ_PERIPH SYSCTL_PERIPH_GPIOJ
```

3.1.2.21 TRF79X0_IRQ_PIN

Definition:
```
#define TRF79X0_IRQ_PIN
```

Description:
Specifies the GPIO pin that is connected to the IRQ signal on the TRF79x0 EM board.

Example: The IRQ signal is on GPIO pin 1.

```
#define TRF79X0_IRQ_PIN GPIO_PIN_1
```

3.1.2.22 TRF79X0_MOD_BASE

Definition:
```
#define TRF79X0_MOD_BASE
```

Description:
Specifies the *base* address of the GPIO port that is connected to the MOD signal on the TRF79x0 EM board.

Example: The MOD signal is on GPIO port J.

```
#define TRF79X0_MOD_BASE GPIO_PORTJ_BASE
```

3.1.2.23 TRF79X0_MOD_PERIPH

Definition:

```
#define TRF79X0_MOD_PERIPH
```

Description:

Specifies the *peripheral* for the GPIO port that is connected to the MOD signal on the TRF79x0 EM board.

Example: The MOD signal is on GPIO port J.

```
#define TRF79X0_MOD_PERIPH SYSCTL_PERIPH_GPIOJ
```

3.1.2.24 TRF79X0_MOD_PIN

Definition:

```
#define TRF79X0_MOD_PIN
```

Description:

Specifies the GPIO pin that is connected to the MOD signal on the TRF79x0 EM board.

Example: The MOD signal is on GPIO pin 4.

```
#define TRF79X0_MOD_PIN GPIO_PIN_4
```

3.1.2.25 TRF79X0_RX_BASE

Definition:

```
#define TRF79X0_RX_BASE
```

Description:

Specifies the *base* address of the GPIO port that is connected to the SSI RX signal on the TRF79x0 EM board.

Example: The SSI peripheral RX signal is on GPIO port A.

```
#define TRF79X0_RX_BASE GPIO_PORTA_BASE
```

3.1.2.26 TRF79X0_RX_CONFIG

Definition:

```
#define TRF79X0_RX_CONFIG
```

Description:

Specifies the GPIO pin that is connected to the SSIRX (DAT1) signal on the TRF79x0 EM board.

Example: The SSI 1 RX signal is on GPIO port A pin 5.

```
#define TRF79X0_RX_CONFIG GPIO_PA5_SSI0XDAT1
```

3.1.2.27 TRF79X0_RX_PERIPH

Definition:

```
#define TRF79X0_RX_PERIPH
```

Description:

Specifies the *peripheral* for the GPIO port that is connected to the SSI RX signal on the TRF79x0 EM board.

Example: The SSI peripheral RX signal is on GPIO port A.

```
#define TRF79X0_RX_PERIPH SYSCTL_PERIPH_GPIOA
```

3.1.2.28 TRF79X0_RX_PIN

Definition:

```
#define TRF79X0_RX_PIN
```

Description:

Specifies the GPIO pin that is connected to the SSI RX signal on the TRF79x0 EM board.

Example: The SSI peripheral RX signal is on GPIO pin 5.

```
#define TRF79X0_RX_PIN GPIO_PIN_5
```

3.1.2.29 TRF79X0_SSI_BASE

Definition:

```
#define TRF79X0_SSI_BASE
```

Description:

Specifies the SSI *base* address for the SSI port that is connected to the TRF79x0 EM board. The value should be set to SYSCTL_PERIPH_SSIn, where n is the number of the SSI port being used.

Example: Uses SSI0 peripheral

```
#define TRF79X0_SSI_BASE SSI0_BASE
```

3.1.2.30 TRF79X0_SSI_PERIPH

Definition:

```
#define TRF79X0_SSI_PERIPH
```

Description:

Specifies the SSI peripheral for the SSI port that is connected to the TRF79x0 EM board. The value should be set to SYSCTL_PERIPH_SSIn, where n is the number of the SSI port being used.

Example: Uses SSI0 peripheral

```
#define TRF79X0_SSI_PERIPH SYSCTL_PERIPH_SSI0
```

3.1.2.31 TRF79X0_TX_BASE

Definition:

```
#define TRF79X0_TX_BASE
```

Description:

Specifies the *base* address of the GPIO port that is connected to the SSI TX signal on the TRF79x0 EM board.

Example: The SSI peripheral TX signal is on GPIO port A.

```
#define TRF79X0_TX_BASE GPIO_PORTA_BASE
```

3.1.2.32 TRF79X0_TX_CONFIG

Definition:

```
#define TRF79X0_TX_CONFIG
```

Description:

Specifies the GPIO pin that is connected to the SSITX (DAT0) signal on the TRF79x0 EM board.

Example: The SSI 1 TX signal is on GPIO port A pin 4.

```
#define TRF79X0_TX_CONFIG GPIO_PA4_SSI0XDAT0
```

3.1.2.33 TRF79X0_TX_PERIPH

Definition:

```
#define TRF79X0_TX_PERIPH
```

Description:

Specifies the *peripheral* for the GPIO port that is connected to the SSI TX signal on the TRF79x0 EM board.

Example: The SSI peripheral TX signal is on GPIO port A.

```
#define TRF79X0_TX_PERIPH SYSCTL_PERIPH_GPIOA
```

3.1.2.34 TRF79X0_TX_PIN

Definition:

```
#define TRF79X0_TX_PIN
```

Description:

Specifies the GPIO pin that is connected to the SSI TX signal on the TRF79x0 EM board.

Example: The SSI peripheral TX signal is on GPIO pin 4.

```
#define TRF79X0_TX_PIN GPIO_PIN_4
```

4 nfclib P2P API

This section covers the NFC API that can be used by an application to access NFC P2P devices.

4.1 NFC P2P API Functions

Data Structures

- sNDEFActionRecord
- sNDEFMessageData
- sNDEFSmartPosterRecord
- sNDEFStatusByte
- sNDEFTextRecord
- sNDEFTextRecordStatusByte
- sNDEFURIRecord
- sNFCP2PRxStatus

Defines

- NDEF_ID_MAXSIZE
- NDEF_STATUSBYTE_CF_CHUNK
- NDEF_STATUSBYTE_GET_CF(ui8x)
- NDEF_STATUSBYTE_GET_IL(ui8x)
- NDEF_STATUSBYTE_GET_MB(ui8x)
- NDEF_STATUSBYTE_GET_ME(ui8x)
- NDEF_STATUSBYTE_GET_SR(ui8x)
- NDEF_STATUSBYTE_GET_TNF(ui8x)
- NDEF_STATUSBYTE_IL_IDLENGTHABSENT
- NDEF_STATUSBYTE_IL_IDLENGTHPRESENT
- NDEF_STATUSBYTE_MB_FIRSTBYTE
- NDEF_STATUSBYTE_ME_LASTBYTE
- NDEF_STATUSBYTE_SET_CF(ui8x)
- NDEF_STATUSBYTE_SET_IL(ui8x)
- NDEF_STATUSBYTE_SET_MB(ui8x)
- NDEF_STATUSBYTE_SET_ME(ui8x)
- NDEF_STATUSBYTE_SET_SR(ui8x)
- NDEF_STATUSBYTE_SET_TNF(ui8x)
- NDEF_STATUSBYTE_SR_1BYTEPAYLOADSIZE
- NDEF_STATUSBYTE_SR_4BYTEPAYLOADSIZE
- NDEF_TEXTRECORD_LANGUAGECODE_MAXSIZE
- NDEF_TEXTRECORD_STATUSBYTE_GET_LENGTHLANGCODE(ui8x)
- NDEF_TEXTRECORD_STATUSBYTE_GET_RFU(ui8x)
- NDEF_TEXTRECORD_STATUSBYTE_GET_UTF(ui8x)

- NDEF_TEXTRECORD_STATUSBYTE_SET_LENGTHLANGCODE(ui8x)
- NDEF_TEXTRECORD_STATUSBYTE_SET_RFU(ui8x)
- NDEF_TEXTRECORD_STATUSBYTE_SET_UTF(ui8x)
- NDEF_TEXTRECORD_STATUSBYTE_UTF16
- NDEF_TEXTRECORD_STATUSBYTE_UTF8
- NDEF_TYPE_ACTION
- NDEF_TYPE_MAXSIZE
- NDEF_TYPE_SIGNATURE
- NDEF_TYPE_SIZE
- NDEF_TYPE_SMARTPOSTER
- NDEF_TYPE_TEXT
- NDEF_TYPE_URI
- NDEF_URIRECORD_IDCODE_RFU

Enumerations

- eNDEF_URIRecord_IDCode
- tAction
- tNFCP2PState
- tTNF

Functions

- sNFCP2PRxStatus NFCP2P_getReceiveState (void)
- void NFCP2P_init (tTRF79x0TRFMode eMode, tTRF79x0Frequency eFrequency)
- bool NFCP2P_NDEFMessageDecoder (sNDEFMessageData ∗psNDEFDataDecoded, uint8_t ∗pui8Buffer, uint16_t ui16BufferMaxLength)
- bool NFCP2P_NDEFMessageEncoder (sNDEFMessageData sNDEFDataToSend, uint8_-t ∗pui8Buffer, uint16_t ui16BufferMaxLength, uint32_t ∗pui32BufferLength)
- bool NFCP2P_NDEFSmartPosterRecordDecoder (sNDEFSmartPosterRecord ∗sSmart-Poster, uint8_t ∗pui8Buffer, uint16_t ui16BufferMaxLength, uint32_t ui32BufferLength)
- bool NFCP2P_NDEFSmartPosterRecordEncoder (sNDEFSmartPosterRecord sSmartPoster, uint8_t ∗pui8Buffer, uint16_t ui16BufferMaxLength, uint32_t ∗pui32BufferLength)
- bool NFCP2P_NDEFTextRecordDecoder (sNDEFTextRecord ∗psTextDataDecoded, uint8_-t ∗pui8Buffer, uint32_t ui32BufferLength)
- bool NFCP2P_NDEFTextRecordEncoder (sNDEFTextRecord sTextRecord, uint8_t ∗pui8Buffer, uint16_t ui16BufferMaxLength, uint32_t ∗pui32BufferLength)
- bool NFCP2P_NDEFURIRecordDecoder (sNDEFURIRecord ∗sURIRecord, uint8_t ∗pui8Buffer, uint32_t ui32BufferLength)
- bool NFCP2P_NDEFURIRecordEncoder (sNDEFURIRecord sURIRecord, uint8_t ∗pui8Buffer, uint16_t ui16BufferMaxLength, uint32_t ∗pui32BufferLength)
- tNFCP2PState NFCP2P_proccessStateMachine (void)
- tStatus NFCP2P_sendPacket (uint8_t ∗pui8DataPtr, uint32_t ui32DataLength)

4.1.1 Detailed Description

This module implements the encoding and decoding of NFC P2P messages and records.

It is assumed that users of this module have a functional knowledge of NFC P2P messages and record types as defined by the NFC specification at `http://www.nfc-forum.org/specs/spec_list`.

The functions in this module assume that the NFCP2P_proccessStateMachine() is being called every 77ms or less as defined by the Digital Protocol Technical Specification requirement 197. Before any of the functions in this module are called, TRF79x0Init() and NFCP2P_init() must be called to initialize the transceiver and the NFCP2P state machine.

4.1.2 Data Structure Documentation

4.1.2.1 sNDEFActionRecord

Definition:

```
typedef struct
{
    tAction eAction;
}
sNDEFActionRecord
```

Members:

eAction Action Record type enumeration.

Description:

This structure defines an Action Record.

4.1.2.2 sNDEFMessageData

Definition:

```
typedef struct
{
    sNDEFStatusByte sStatusByte;
    uint8_t ui8TypeLength;
    uint32_t ui32PayloadLength;
    uint8_t ui8IDLength;
    uint8_t pui8Type[NDEF_TYPE_MAXSIZE];
    uint8_t pui8ID[NDEF_ID_MAXSIZE];
    uint8_t *pui8PayloadPtr;
}
sNDEFMessageData
```

Members:

sStatusByte Metadata about the message.

ui8TypeLength Length of the Type field in bytes.

ui32PayloadLength Length of the payload in bytes.

ui8IDLength Length of ID field in bytes. Optional field.

pui8Type Contains message type.

pui8ID Contains message ID. Optional field.

pui8PayloadPtr Pointer to the encoded payload buffer.

Description:

Structure to hold NDEF Message header data. The message header encapsulates and contains metadata about the payload message. This structure is used with the NFCP2P_NDEFMessageEncoder and NFCP2P_NDEFMessageDecoder functions. For detailed information on the NDEF message header data, please see the NFC specification.

4.1.2.3 sNDEFSmartPosterRecord

Definition:

```
typedef struct
{
    sNDEFMessageData sTextHeader;
    sNDEFTextRecord sTextPayload;
    sNDEFMessageData sURIHeader;
    sNDEFURIRecord sURIPayload;
    bool bActionExists;
    sNDEFMessageData sActionHeader;
    sNDEFActionRecord sActionPayload;
}
sNDEFSmartPosterRecord
```

Members:

sTextHeader message header for Text Record

sTextPayload Text Record payload structure.

sURIHeader message header for URI Record

sURIPayload URI Record payload strucutre.

bActionExists Flag to signal if Action Record is part of Smart Poster.

sActionHeader message header for Action Record

sActionPayload Action Record payload strucutre.

Description:

This structure defines the SmartPoster record type. The SmartPoster Record is essentially a URI Record with other records included for metadata. Thus the SmartPoster must include at least a URI Record and may also include a Text Record for a Title record, an Action record to do actions on the URI, an Icon Record with a small icon, a Size record that holds the size of the externally referenced entity, and a Type record that denotes the type of the externally referenced entity. It should be noted that while the SmartPoster specification can include all these records, this library only provides support for Title, URI and Action records. All other records are ignored by the default handler.

4.1.2.4 sNDEFStatusByte

Definition:

```
typedef struct
{
    bool MB;
    bool ME;
    bool CF;
    bool SR;
    bool IL;
    tTNF TNF;
}
sNDEFStatusByte
```

Members:

MB Message Begin flag.

ME Message End flag.

CF Chunk Flag.

SR Short Record flag.

IL ID Length flag.

TNF Type Name Field. An enumeration specifying the general tag type.

Description:

NFC NDEF message header StatusByte structure. Included in this structure are fields for Message Begin (MB), Message End (ME), Chunk Flag (CF), Short Record (SR), IDLength (IL) and Type Name Format (TNF). The purpose of this structure is to make the fields readily available for message processing.

4.1.2.5 sNDEFTextRecord

Definition:

```
typedef struct
{
    sNDEFTextRecordStatusByte sStatusByte;
    uint8_t pui8LanguageCode[NDEF_TEXTRECORD_LANGUAGECODE_MAXSIZE];
    uint8_t *pui8Text;
    uint32_t ui32TextLength;
}
sNDEFTextRecord
```

Members:

sStatusByte Structure to hold StatusByte information.

pui8LanguageCode Buffer that holds the Language Code.

pui8Text Pointer to the Text Buffer.

ui32TextLength Length of text in Text Buffer.

Description:

This structure defines the text record. sStatusByte contains the length of the language code and the formatting for the Text (UTF8/UTF16). pui8LanguageCode is a buffer that contains the language code; the buffer size can be changed at compile time by modifying the NDEF_TEXTRECORD_LANGUAGECODE_MAXSIZE define. pui8Text is a pointer to the text payload of the Text Record. These three fields are defined in the NFC specification. In addition, ui32TextLength has been added for convenience to keep track of the Text buffer length. For example, a text record with the Text "hello world" would have a StatusByte of 0x02 (UTF = 0 (UTF8), LenLangCode = 0x2), a Language Code of "en" (for English, note that it is 2 bytes long just as the ui5LengthLangCode field of the Text Record StatusByte denoted), pui8Text points to a buffer holding "hello world", and ui32TextLength has a value of 11, which is the number of chars in "hello world".

4.1.2.6 sNDEFTextRecordStatusByte

Definition:

```
typedef struct
{
    bool bUTFcode;
    bool bRFU;
    uint8_t ui5LengthLangCode;
}
sNDEFTextRecordStatusByte
```

Members:

bUTFcode Flag for UTF Code. 0 = UTF8, 1 = UTF16.

bRFU Reserved for future use by NFC specification.

ui5LengthLangCode Length of Text Record language code.

Description:

This structure defines the text record status byte. bUTFcode determines if the Text Record is encoded with UTF8 (0) or UTF16 (1). bRFU is reserved for future use by the NFC specification. ui5LengthLangCode holds the length of the language code. Currently language code lengths are either 2 or 5 bytes.

4.1.2.7 sNDEFURIRecord

Definition:

```
typedef struct
{
    eNDEF_URIRecord_IDCode eIDCode;
    uint8_t *puiUTF8String;
    uint32_t ui32URILength;
}
sNDEFURIRecord
```

Members:

eIDCode Enumeration of all possible ID codes.

puiUTF8String Buffer that holds the URI character string.

ui32URILength Length of URI Character String.

Description:

This structure defines the URI record type. The URI Record Type has two fields; the ID code and the UTF8 URI string. The IDCode is used to determine the URI type. For example, IDcode of 0x06 is 'mailto:' and usually triggers an email event. IDcode 0x01 is 'http://www.' and usually triggers a webpage to open. The IDcode values are prepended to the UTF8 string. ui32URILength is used to determine the length of the puiUTF8String buffer. For example, to direct a user to 'http://www.ti.com' the IDcode is 0x01, the UTF8 string is 'ti.com', and the ui32URILength is 0x6.

4.1.2.8 sNFCP2PRxStatus

Definition:

```
typedef struct
{
    tPacketStatus eDataReceivedStatus;
    uint8_t ui8DataReceivedLength;
    uint8_t *pui8RxDataPtr;
}
sNFCP2PRxStatus
```

Members:

eDataReceivedStatus SNEP RX Packet Status.

ui8DataReceivedLength SNEP Number of bytes received.

pui8RxDataPtr Pointer to data received.

Description:

This structure defines the status of the received payload.

4.1.3 Define Documentation

4.1.3.1 NDEF_ID_MAXSIZE

Definition:

```
#define NDEF_ID_MAXSIZE
```

Description:

Maximum size of the ID field in StatusByte, which can be modified to support larger ID names.

Example: Copy the ID from the raw buffer to the structure using NDEF_ID_MAXSIZE to prevent overflowing buffer in the structure

```
//
// Assume IDLength already decoded from the raw buffer is stored in
// sNDEFMessageData.ui8IDLength. Assume ui8RawBuffer is a pointer to
// the beginning of the ID field in the raw data stream.
//
int x = 0;
for(x = 0; (x<NDEF_ID_MAXSIZE) & (x<sNDEFMessageData.ui8IDLength); x++)
{
    sNDEFMessageData.pui8pui8ID[x] = ui8RawBuffer[x];
}
```

4.1.3.2 NDEF_STATUSBYTE_CF_CHUNK

Definition:

```
#define NDEF_STATUSBYTE_CF_CHUNK
```

Description:

Flag used to check the CF field in the StatusByte. If CF is set, then the message is a chunked message spread out across multiple transactions.

Example: Check for Chunked Flag

```
if(NDEF_STATUSBYTE_GET_CF(ui8StatusByte) == NDEF_STATUSBYTE_CF_CHUNK)
{}
```

4.1.3.3 NDEF_STATUSBYTE_GET_CF

Macro used to get the CF field value from the StatusByte of the NFC message header.

Definition:

```
#define NDEF_STATUSBYTE_GET_CF(ui8x)
```

Parameters:

ui8x is the 8-bit StatusByte

Description:

Example: Get the CF field from the StatusByte into variable x

```
x = NDEF_STATUSBYTE_GET_CF(sNDEFMessageData.sStatusByte)
```

4.1.3.4 #define NDEF_STATUSBYTE_GET_IL(ui8x) ((ui8x >> 3) & 0x01)

Macro used to get the IL field value from the StatusByte of the NFC message header.

Parameters:

ui8x is the 8-bit StatusByte

Example: Get the IL field from the StatusByte into variable x

```
x = NDEF_STATUSBYTE_GET_IL(sNDEFMessageData.sStatusByte)
```

4.1.3.5 NDEF_STATUSBYTE_GET_MB

Macro used to get the MB field value from the StatusByte of the NFC message header.

Definition:

```
#define NDEF_STATUSBYTE_GET_MB(ui8x)
```

Parameters:

ui8x is the 8-bit StatusByte

Description:

Example: Get the MB field from the StatusByte into variable x

```
x = NDEF_STATUSBYTE_GET_MB(sNDEFMessageData.sStatusByte)
```

4.1.3.6 #define NDEF_STATUSBYTE_GET_ME(ui8x) ((ui8x >> 6) & 0x01)

Macro used to get the ME field value from the StatusByte of the NFC message header.

Parameters:

ui8x is the 8-bit StatusByte

Example: Get the ME field from the StatusByte into variable x

```
x = NDEF_STATUSBYTE_GET_ME(sNDEFMessageData.sStatusByte)
```

4.1.3.7 NDEF_STATUSBYTE_GET_SR

Macro used to get the SR field value from the StatusByte of the NFC message header.

Definition:
```
#define NDEF_STATUSBYTE_GET_SR(ui8x)
```

Parameters:
ui8x is the 8-bit StatusByte

Description:
Example: Get the SR field from the StatusByte into variable x

```
x = NDEF_STATUSBYTE_GET_SR(sNDEFMessageData.sStatusByte)
```

4.1.3.8 #define NDEF_STATUSBYTE_GET_TNF(ui8x) ((ui8x >> 0) & 0x07)

Macro used to get the TNF field value from the StatusByte of the NFC message header.

Parameters:
ui8x is the 8-bit StatusByte

Example: Get the TNF field from the StatusByte into variable x

```
x = NDEF_STATUSBYTE_GET_TNF(sNDEFMessageData.sStatusByte)
```

4.1.3.9 NDEF_STATUSBYTE_IL_IDLENGTHABSENT

Definition:
```
#define NDEF_STATUSBYTE_IL_IDLENGTHABSENT
```

Description:
Flag used to check the IL field in the StatusByte. If IL is not set, then there is no ID or IDLength fields included in the message.

Example: Check for the presence of the ID Length and ID name field

```
if(NDEF_STATUSBYTE_GET_IL(ui8StatusByte) == NDEF_STATUSBYTE_IL_-
IDLENGTHABSENT) {}
```

4.1.3.10 NDEF_STATUSBYTE_IL_IDLENGTHPRESENT

Definition:
```
#define NDEF_STATUSBYTE_IL_IDLENGTHPRESENT
```

Description:
Flag used to check the IL field in the StatusByte. If IL is set, then the ID and IDLength fields are present in the message.

Example: Check for the presence of the ID Length and ID name field

```
if(NDEF_STATUSBYTE_GET_IL(ui8StatusByte) == NDEF_STATUSBYTE_IL_-
IDLENGTHPRESENT) {}
```

4.1.3.11 NDEF_STATUSBYTE_MB_FIRSTBYTE

Definition:

```
#define NDEF_STATUSBYTE_MB_FIRSTBYTE
```

Description:

Flag used to check the MB field in the StatusByte. If MB is set then this is the first Record.

Example: Check for Message Begin flag

```
if(NDEF_STATUSBYTE_GET_MB(ui8StatusByte) == NDEF_STATUSBYTE_MB_-
FIRSTBYTE){}
```

4.1.3.12 NDEF_STATUSBYTE_ME_LASTBYTE

Definition:

```
#define NDEF_STATUSBYTE_ME_LASTBYTE
```

Description:

Flag used to check the ME field in the StatusByte. If ME is set, then this is the last Record.

Example: Check for Message End flag

```
if(NDEF_STATUSBYTE_GET_ME(ui8StatusByte) == NDEF_STATUSBYTE_ME_-
LASTBYTE) {}
```

4.1.3.13 NDEF_STATUSBYTE_SET_CF

This Macro is used to set the CF field in the StatusByte of the NFC message header by shifting a bit into position. This define should be ORed together with other StatusByte Fields.

Definition:

```
#define NDEF_STATUSBYTE_SET_CF(ui8x)
```

Parameters:

ui8x is the binary value to be shifted into place

Description:

Example: Set the CF field in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte | NDEF_-
STATUSBYTE_SET_CF(0x1)
```

Example: Clear the CF field in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte & NDEF_-
STATUSBYTE_SET_CF(0x0)
```

4.1.3.14 #define NDEF_STATUSBYTE_SET_IL(ui8x) ((ui8x & 0x01) << 3)

This macro is used to set the IL field in the StatusByte of the NFC message header by shifting a bit into position. This define should be ORed together with other StatusByte Fields.

Parameters:
ui8x is the binary value to be shifted into place

Example: Set the IL field in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte | NDEF_STATUSBYTE_-
SET_IL(0x1)
```

Example: Clear the IL field in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte & NDEF_STATUSBYTE_-
SET_IL(0x0)
```

4.1.3.15 NDEF_STATUSBYTE_SET_MB

This macro is used to set the MB field in the StatusByte of the NFC message header by shifting a bit into position. This define should be ORed together with other StatusByte Fields.

Definition:
```
#define NDEF_STATUSBYTE_SET_MB(ui8x)
```

Parameters:
ui8x is the binary value to be shifted into place

Description:
Example: Set the MB field in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte | NDEF_-
STATUSBYTE_SET_MB(0x1)
```

Example: Clear the MB field in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte & NDEF_-
STATUSBYTE_SET_MB(0x0)
```

4.1.3.16 #define NDEF_STATUSBYTE_SET_ME(ui8x) ((ui8x & 0x01) << 6)

This macro is used to set the ME field in the StatusByte of the NFC message header by shifting a bit into position. This define should be ORed together with other StatusByte Fields.

Parameters:
ui8x is the binary value to be shifted into place

Example: Set the ME field in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte | NDEF_STATUSBYTE_-
SET_ME(0x1)
```

Example: Clear the ME field in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte & NDEF_STATUSBYTE_-
SET_ME(0x0)
```

4.1.3.17 NDEF_STATUSBYTE_SET_SR

This macro is used to set the SR field in the StatusByte of the NFC message header by shifting a bit into position. This define should be ORed together with other StatusByte Fields.

Definition:

```
#define NDEF_STATUSBYTE_SET_SR(ui8x)
```

Parameters:

ui8x is the binary value to be shifted into place

Description:

Example: Set the SR field in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte | NDEF_-
STATUSBYTE_SET_SR(0x1)
```

Example: Clear the SR field in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte & NDEF_-
STATUSBYTE_SET_SR(0x0)
```

4.1.3.18 #define NDEF_STATUSBYTE_SET_TNF(ui8x) ((ui8x & 0x07) << 0)

This macro is used to set the TNF field in the StatusByte of the NFC message header by shifting a bit into position. This define should be ORed together with other StatusByte Fields.

Parameters:

ui8x is the 3-bit value to be shifted into place

Example: Set the TNF field to Well Known Type in a StatusByte

```
NsNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte | NDEF_STATUSBYTE_-
SET_TNF(0x1)
```

Example: Set the TNF field to Unknown Type in a StatusByte

```
sNDEFMessage.sStatusByte = sNDEFMessage.sStatusByte & NDEF_STATUSBYTE_-
SET_TNF(0x5)
```

4.1.3.19 NDEF_STATUSBYTE_SR_1BYTEPAYLOADSIZE

Definition:

```
#define NDEF_STATUSBYTE_SR_1BYTEPAYLOADSIZE
```

Description:

Flag used to check the SR field in the StatusByte. If SR is set, then the message is a short record with a payload length field of 1 byte instead of 4 bytes.

Example: Check the Short Record flag

```
if(NDEF_STATUSBYTE_GET_SR(ui8StatusByte) == NDEF_STATUSBYTE_SR_-
1BYTEPAYLOADSIZE){}
```

4.1.3.20 NDEF_STATUSBYTE_SR_4BYTEPAYLOADSIZE

Definition:

```
#define NDEF_STATUSBYTE_SR_4BYTEPAYLOADSIZE
```

Description:

Flag used to check the SR field in the StatusByte. If SR is not set, then the message is a normal record with a payload length field of 4 bytes instead of 1 byte.

Example: Check the Short Record flag

```
if(NDEF_STATUSBYTE_GET_SR(ui8StatusByte) == NDEF_STATUSBYTE_SR_-
4BYTEPAYLOADSIZE){}
```

4.1.3.21 NDEF_TEXTRECORD_LANGUAGECODE_MAXSIZE

Definition:

```
#define NDEF_TEXTRECORD_LANGUAGECODE_MAXSIZE
```

Description:

Define the size of the Text Record Language Code Buffer. This can be changed by the user to fit larger language codes that may develop in the future. Current language codes are 2 or 5 bits, but users can use larger sizes if they are adopted in the future.

4.1.3.22 NDEF_TEXTRECORD_STATUSBYTE_GET_LENGTHLANGCODE

This macro extracts the Language Code Length field from the raw StatusByte.

Definition:

```
#define NDEF_TEXTRECORD_STATUSBYTE_GET_LENGTHLANGCODE(ui8x)
```

Parameters:

ui8x is the 8-bit StatusByte

Description:

Example: Fill the Language Code Length field in the data structure from the raw buffer byte

```
sNDEFTextRecord.ui5LengthLangCode = NDEF_TEXTRECORD_STATUSBYTE_GET_-
LENGTHLANGCODE(ui8StatusByte)
```

4.1.3.23 #define NDEF_TEXTRECORD_STATUSBYTE_GET_RFU(ui8x) ((ui8x >> 6) & 0x01)

This macro extracts the RFU bit value from raw StatusByte. According to the NFC specification, this value must be zero.

Parameters:

ui8x is the 8-bit StatusByte

Example: Fill the RFU boolean value in the data structure from the raw buffer byte

```
sNDEFTextRecord.bRFU = NDEF_TEXTRECORD_STATUSBYTE_GET_RFU( ui8Status-
Byte)
```

4.1.3.24 NDEF_TEXTRECORD_STATUSBYTE_GET_UTF

This macro extracts the UTF bit value from the raw StatusByte.

Definition:

```
#define NDEF_TEXTRECORD_STATUSBYTE_GET_UTF(ui8x)
```

Parameters:

ui8x is the 8-bit StatusByte

Description:

Example: Fill the UTF boolean value in the data structure from the raw buffer byte

```
sNDEFTextRecord.bUTFcode = NDEF_TEXTRECORD_STATUSBYTE_GET_UTF(
ui8StatusByte)
```

4.1.3.25 #define NDEF_TEXTRECORD_STATUSBYTE_SET_-LENGTHLANGCODE(ui8x) ((ui8x & 0x3F) << 0)

Set the Language Code Length field in the TextRecord StatusByte field. This define should be ORed together with other StatusByte fields and set into StatusByte.

Parameters:

ui8x is the 8-bit StatusByte

```
ui8StatusByte = (NDEF_TEXTRECORD_STATUSBYTE_SET_LENGTHLANGCODE(5) |
NDEF_TEXTRECORD_STATUSBYTE_SET_LENGTHLANGCODE(...))
```

4.1.3.26 NDEF_TEXTRECORD_STATUSBYTE_SET_RFU

Set the RFU bit field in the TextRecord StatusByte field. Should be ORed together with other StatusByte fields and set into StatusByte. The RFU field is reserved for future use by the NFC specification and should not be used by normal applications.

Definition:

```
#define NDEF_TEXTRECORD_STATUSBYTE_SET_RFU(ui8x)
```

Parameters:

ui8x is the 8-bit StatusByte

Description:

```
ui8StatusByte = (NDEF_TEXTRECORD_STATUSBYTE_SET_RFU(0)| (NDEF_-
TEXTRECORD_STATUSBYTE_SET_LENGTHLANGCODE(...)))
```

4.1.3.27 #define NDEF_TEXTRECORD_STATUSBYTE_SET_UTF(ui8x) ((ui8x & 0x01) << 7)

Set UTF bit field in TextRecord StatusByte field. This define should be ORed together with other StatusByte fields and set into StatusByte.

Parameters:
ui8x is the 8-bit StatusByte

Example: Set UTF bit field to UTF8

```
ui8StatusByte = (NDEF_TEXTRECORD_STATUSBYTE_SET_UTF( NDEF_TEXTRECORD_-
STATUSBYTE_UTF8) | NDEF_TEXTRECORD_STATUSBYTE_SET_LENGTHLANGCODE(...))
```

4.1.3.28 NDEF_TEXTRECORD_STATUSBYTE_UTF16

Definition:

```
#define NDEF_TEXTRECORD_STATUSBYTE_UTF16
```

Description:
Check text record bit in the StatusByte to determine if text record is UTF16 format.

Example: Check Text Record for UTF16 format

```
if(sNDEFTextRecord.bUTFcode == NDEF_TEXTRECORD_STATUSBYTE_UTF16){}
```

4.1.3.29 NDEF_TEXTRECORD_STATUSBYTE_UTF8

Definition:

```
#define NDEF_TEXTRECORD_STATUSBYTE_UTF8
```

Description:
Check text record bit in the StatusByte to determine if text record is UTF8 format.

Example: Check Text Record for UTF8 format

```
if(sNDEFTextRecord.bUTFcode == NDEF_TEXTRECORD_STATUSBYTE_UTF8){}
```

4.1.3.30 NDEF_TYPE_ACTION

Definition:

```
#define NDEF_TYPE_ACTION
```

Description:
NFC Message TypeID hex representation for ACTION records. 0x616374 == "act" in UTF-8

Example: Check if tag type is ACTION

```
//
// Assume the Tag Type has already decoded into sNDEFMessageData.pui8Type
//
if(sNDEFMessageData.pui8Type == NDEF_TYPE_ACTION)
{
    // The Tag is a ACTION record, handle it appropriately
}
```

4.1.3.31 NDEF_TYPE_MAXSIZE

Definition:

`#define NDEF_TYPE_MAXSIZE`

Description:

Maximum size of Type field in StatusByte. This define is used to declare the length of the buffer in the structure and thus can be changed to allow larger Type names.

Example: Copy the Type from raw buffer to structure using NDEF_TYPE_MAXSIZE to prevent overflowing buffer in the structure

```
//
// Assume that TypeLength is already decoded from the raw buffer and is
// stored in sNDEFMessageData.ui8TypeLength. Assume ui8RawBuffer is a
// pointer to the beginning of the Type field in the raw data stream.
//
int x = 0;
for(x = 0; (x<NDEF_TYPE_MAXSIZE) & (x<sNDEFMessageData.ui8TypeLength); x++)
{
    sNDEFMessageData.pui8Type[x]=ui8RawBuffer[x];
}
```

4.1.3.32 NDEF_TYPE_SIGNATURE

Definition:

`#define NDEF_TYPE_SIGNATURE`

Description:

NFC Message TypeID hex representation for SIGNATURE records. 0x536967 == "Sig" in UTF-8

Example: Check if tag type is SIGNATURE

```
//
// Assume the Tag Type has already decoded into sNDEFMessageData.pui8Type
//
if(sNDEFMessageData.pui8Type == NDEF_TYPE_SIGNATURE)
{
    // The Tag is a SIGNATURE record, handle it appropriately
}
```

4.1.3.33 NDEF_TYPE_SIZE

Definition:

`#define NDEF_TYPE_SIZE`

Description:

NFC Message TypeID hex representation for SIZE records. 0x73 == 's' in UTF-8

Example: Check if tag type is SIZE

```
//
// Assume the Tag Type has already decoded into sNDEFMessageData.pui8Type
//
if(sNDEFMessageData.pui8Type == NDEF_TYPE_SIZE)
```

```
{
    // The Tag is a SIZE record. Handle it appropriately
}
```

4.1.3.34 NDEF_TYPE_SMARTPOSTER

Definition:

```
#define NDEF_TYPE_SMARTPOSTER
```

Description:

NFC Message TypeID hex representation for SMARTPOSTER records. 0x5370 == "Sp" in UTF-8

Example: Check if tag type is SMARTPOSTER

```
//
// Assume the Tag Type has already decoded into sNDEFMessageData.pui8Type
//
if(sNDEFMessageData.pui8Type == NDEF_TYPE_SMARTPOSTER)
{
    // The Tag is a SMARTPOSTER record, handle it appropriately
}
```

4.1.3.35 NDEF_TYPE_TEXT

Definition:

```
#define NDEF_TYPE_TEXT
```

Description:

NFC Message TypeID hex representation for TEXT records. 0x54 == ’T’ in UTF-8

Example: Check if tag type is TEXT

```
//
// Assume the Tag Type has already decoded into sNDEFMessageData.pui8Type
//
if(sNDEFMessageData.pui8Type == NDEF_TYPE_TEXT)
{
    // The Tag is a TEXT record, handle it appropriately
}
```

4.1.3.36 NDEF_TYPE_URI

Definition:

```
#define NDEF_TYPE_URI
```

Description:

NFC Message TypeID hex representation for URI records. 0x55 == ’U’ in UTF-8

Example: Check if tag type is URI

```
//
// Assume the Tag Type has already decoded into sNDEFMessageData.pui8Type
//
```

```
if(sNDEFMessageData.pui8Type == NDEF_TYPE_URI)
{
    // The Tag is a URI record, handle it appropriately
}
```

4.1.3.37 NDEF_URIRECORD_IDCODE_RFU

Definition:

`#define NDEF_URIRECORD_IDCODE_RFU`

Description:

Define used to mark end of well-defined URI Record ID Codes. Any code greater than this value is not defined by the NFC specification.

Example: Check if ID Code of Tag is known defined value

```
if(sNDEFURIRecord.eIDCode < NDEF_URIRECORD_IDCODE_RFU)
        {
            //process tag
        }
```

4.1.4 Enumeration Documentation

4.1.4.1 eNDEF_URIRecord_IDCode

Description:

Enumeration of all possible URI Record ID Codes defined by the NFC specification. For the complete list, please see the enumeration definition in nfc_p2p.h. Defined values range from 0x00 (no prepending) to 0x23 ('urn:nfc:'). Values 0x24 and above are reserved for future use.

Enumerators:

unabridged Nothing is prepended to puiUTF8String.
http_www 'http://www.' is prepended to puiUTF8String
https_www 'https://www.' is prepended to puiUTF8String
http 'http://' is prepended to puiUTF8String
https 'https://' is prepended to puiUTF8String
tel 'tel:' is prepended to puiUTF8String
mailto 'mailto:' is prepended to puiUTF8String
ftp_anonymous 'ftp://anonymous:anonymous@' is prepended to puiUTF8String
ftp_ftp 'ftp://ftp.' is prepended to puiUTF8String
ftps 'ftps://' is prepended to puiUTF8String
sftp 'sftp://' is prepended to puiUTF8String
smb 'smb://' is prepended to puiUTF8String
nfs 'nfs://' is prepended to puiUTF8String
ftp 'ftp://' is prepended to puiUTF8String
dav 'dav://' is prepended to puiUTF8String
news 'news:' is prepended to puiUTF8String
telnet 'telnet://' is prepended to puiUTF8String
imap 'imap:' is prepended to puiUTF8String

rtsp 'rtsp://' is prepended to puiUTF8String
urn 'urn:' is prepended to puiUTF8String
pop 'pop:' is prepended to puiUTF8String
sip 'sip:' is prepended to puiUTF8String
sips 'sips:' is prepended to puiUTF8String
tftp 'tftp:' is prepended to puiUTF8String
btspp 'btspp://' is prepended to puiUTF8String
btl2cap 'btl2cap://' is prepended to puiUTF8String
btgoep 'btgoep://' is prepended to puiUTF8String
tcpobex 'tcpobex://' is prepended to puiUTF8String
irdaobex 'irdaobex://' is prepended to puiUTF8String
file 'file://' is prepended to puiUTF8String
urn_epc_id 'urn:epc:id:' is prepended to puiUTF8String
urn_epc_tag 'urn:epc:tag:' is prepended to puiUTF8String
urn_epc_pat 'urn:epc:pat:' is prepended to puiUTF8String
urn_epc_raw 'urn:epc:raw:' is prepended to puiUTF8String
urn_epc 'urn:epc:' is prepended to puiUTF8String
urn_nfc 'urn:nfc:' is prepended to puiUTF8String
RFU Values equal to and above this are reserved for future use (RFU).

4.1.4.2 tAction

Description:
Enumeration of the three actions that can be associated with an Action Record.

Enumerators:
DO_ACTION Do Action on Record.
SAVE_FOR_LATER Save Record for Later.
OPEN_FOR_EDITING Open Record for Editing.

4.1.4.3 tNFCP2PState

Description:
Enumeration for 4 possible states for NFC P2P State Machine.

Enumerators:
NFC_P2P_PROTOCOL_ACTIVATION Polling/Listening for SENSF_REQ / SENSF_RES.
NFC_P2P_PARAMETER_SELECTION Setting the NFCIDs and bit rate.
NFC_P2P_DATA_EXCHANGE_PROTOCOL Data exchange using the LLCP layer.
NFC_P2P_DEACTIVATION Technology deactivation.

4.1.4.4 tTNF

Description:
Enumeration for Type Name Format (TNF) field in NDEF header StatusByte. TNF values are 3 bits. Most records are of the Well Known Type format (0x01).

Enumerators:

TNF_EMPTY Empty Format.

TNF_WELLKNOWNTYPE NFC Forum Well Known Type [NFC RTD].

TNF_MEDIA_TYPE Media-type as defined in RFC 2046 [RFC 2046].

TNF_ABSOLUTE_URI Absolute URI as defined in RFC 3986 [RFC 3986].

TNF_EXTERNAL_TYPE NFC Forum external type [NFC RTD].

TNF_UNKNOWN Unknown.

TNF_UNCHANGED Unchanged (used with single message across multiple chunks).

TNF_RESERVED Reserved.

4.1.5 Function Documentation

4.1.5.1 NFCP2P_getReceiveState

NFCP2P_getReceiveState - Gets the receive state from the low level SNEP stack.

Prototype:

```
sNFCP2PRxStatus
NFCP2P_getReceiveState(void)
```

Description:

Description: This function is used to get the receive payload status from the SNEP layer.

Returns:

This function returns the receive state.

4.1.5.2 NFCP2P_init

Initialize the variables used by the NFC Stack.

Prototype:

```
void
NFCP2P_init(tTRF79x0TRFMode eMode,
            tTRF79x0Frequency eFrequency)
```

Parameters:

eMode is the mode which to initialize the TRF79x0

eFrequency is the frequency which to initialize the TRF79x0

Description:

This function must be called before any other NFCP2P function is called. It can be called at any point to change the mode or frequency of the TRF79x0 transceiver. This function initializes either the initiator or the target mode.

The *eMode* parameter can be any of the following:

- **BOARD_INIT** - Initial Mode.
- **P2P_INITATIOR_MODE** - P2P Initiator Mode.
- **P2P_PASSIVE_TARGET_MODE** - P2P Passive Target Mode.

- **P2P_ACTIVE_TARGET_MODE** - P2P Active Target Mode.
- **CARD_EMULATION_TYPE_A** - Card Emulation for Type A cards.
- **CARD_EMULATION_TYPE_B** - Card Emulation for Type B cards.

The *eFrequency* parameter can be any of the following:

- **FREQ_STAND_BY** - Used for Board Initialization.
- **FREQ_106_KBPS** - Frequency of 106 kB per second.
- **FREQ_212_KBPS** - Frequency of 212 kB per second.
- **FREQ_424_KBPS** - Frequency of 424 kB per second.

Returns:
None.

4.1.5.3 NFCP2P_NDEFMessageDecoder

Decodes NFC Message meta-data and payload information.

Prototype:
```
bool
NFCP2P_NDEFMessageDecoder(sNDEFMessageData *psNDEFDataDecoded,
                          uint8_t *pui8Buffer,
                          uint16_t ui16BufferMaxLength)
```

Parameters:
psNDEFDataDecoded is a pointer to the sNDEFMessageData structure to be filled.
pui8Buffer is a pointer to the raw NFC data buffer from which to decode the data.
ui16BufferMaxLength is the maximum number of bytes the buffer can hold. This parameter is used to prevent reading past the end of the buffer.

Description:
This function takes in a buffer of raw NFC data and fills up an sNDEFMessageData structure. This function is the first step to decoding an NFC Message. The next step is to decode the Message Payload, which is the record. The decoded sNDEFMessageData structure has a field named **pui8Type**. The **pui8Type** field defines the record type and therefore indicates which RecordDecoder function to use on the Message Payload.

Returns:
This function returns **STATUS_SUCCESS** (1) or **STATUS_FAIL** (0).

4.1.5.4 NFCP2P_NDEFMessageEncoder

Encodes NFC Message meta-data and payload information.

Prototype:
```
bool
NFCP2P_NDEFMessageEncoder(sNDEFMessageData sNDEFDataToSend,
                          uint8_t *pui8Buffer,
                          uint16_t ui16BufferMaxLength,
                          uint32_t *pui32BufferLength)
```

Parameters:

sNDEFDataToSend is a sNDEFMessageData structure filled out with the NDEF message to send.

pui8Buffer is a pointer to the buffer where the raw encoded data will be stored

ui16BufferMaxLength is the maximum number of bytes the buffer can hold. This parameter is used to prevent writing past the end of the buffer.

pui32BufferLength is a pointer to an integer that is filled with the length of the raw data encoded to the **pui8Buffer**.

Description:

This function takes a filled sNDEFMessageData structure and encodes it to the provided buffer. The length, in bytes, of the data encoded to the buffer is stored into the integer pointer provided.

Returns:

This function returns **STATUS_SUCCESS** (1) or **STATUS_FAIL** (0).

4.1.5.5 NFCP2P_NDEFSmartPosterRecordDecoder

Decode NDEF SmartPoster Records.

Prototype:

```
bool
NFCP2P_NDEFSmartPosterRecordDecoder(sNDEFSmartPosterRecord
*sSmartPoster,
                                    uint8_t *pui8Buffer,
                                    uint16_t ui16BufferMaxLength,
                                    uint32_t ui32BufferLength)
```

Parameters:

sSmartPoster is a pointer to the SmartPosterRecord structure into which to decode the data.

pui8Buffer is a pointer to the raw NFC data buffer to be decoded.

ui16BufferMaxLength is the maximum number of bytes the buffer can hold. This parameter is used to prevent reading past the end of the buffer.

ui32BufferLength is the length of the raw NFC data buffer.

Description:

This function takes a raw NFC data buffer and decodes the data into a SmartPoster record data structure. It is assumed that the raw data buffer contains a SmartPoster record.

Returns:

This function returns **STATUS_SUCCESS** (1) or **STATUS_FAIL** (0).

Note:

Currently only Title, Action and URI records are supported. Other records are skipped and ignored.

4.1.5.6 NFCP2P_NDEFSmartPosterRecordEncoder

Encode NDEF SmartPoster Records.

Prototype:

```
bool
NFCP2P_NDEFSmartPosterRecordEncoder(sNDEFSmartPosterRecord
sSmartPoster,
                                    uint8_t *pui8Buffer,
                                    uint16_t ui16BufferMaxLength,
                                    uint32_t *pui32BufferLength)
```

Parameters:

sSmartPoster is the SmartPoster Record Structure to be encoded.

pui8Buffer is a pointer to the buffer to fill with the raw NFC data.

ui16BufferMaxLength is the maximum number of bytes the buffer can hold. This parameter is used to prevent writing past the end of the buffer.

pui32BufferLength is a pointer to the integer to hold the length of the raw NFC data buffer.

Description:

This function takes a SmartPoster record structure and encodes it into a provided buffer in a raw NFC data format. The length of the data stored in the buffer is stored in **pui32BufferLength**.

Note:

It is assumed that all smart poster messages have a Text record and a URI record.

Returns:

This function returns **STATUS_SUCCESS** (1) or **STATUS_FAIL** (0).

4.1.5.7 NFCP2P_NDEFTextRecordDecoder

Decode NDEF Text Records.

Prototype:

```
bool
NFCP2P_NDEFTextRecordDecoder(sNDEFTextRecord *psTextDataDecoded,
                             uint8_t *pui8Buffer,
                             uint32_t ui32BufferLength)
```

Parameters:

psTextDataDecoded is a pointer to the TextRecord structure to decode the data into.

pui8Buffer is a pointer to the raw NFC data buffer to be decoded.

ui32BufferLength is the length of the raw NFC data buffer.

Description:

This function takes a raw NFC data buffer and decodes the data into a Text record data structure. It is assumed that the raw data buffer contains a text record.

Returns:

This function returns **STATUS_SUCCESS** (1) or **STATUS_FAIL** (0).

4.1.5.8 NFCP2P_NDEFTextRecordEncoder

Encode NDEF Text Records.

Prototype:

```
bool
NFCP2P_NDEFTextRecordEncoder(sNDEFTextRecord sTextRecord,
                             uint8_t *pui8Buffer,
                             uint16_t ui16BufferMaxLength,
                             uint32_t *pui32BufferLength)
```

Parameters:

sTextRecord is the Text Record Structure to be encoded.

pui8Buffer is a pointer to the buffer to fill with the raw NFC data.

ui16BufferMaxLength is the maximum number of bytes the buffer can hold. This parameter is used to prevent writing past the end of the buffer.

pui32BufferLength is a pointer to the integer to hold the length of the raw NFC data buffer.

Description:

This function takes a TextRecord structure and encodes it into a provided buffer in the raw NFC data format. The length of the data stored in the buffer is stored in *ui32BufferLength*.

Returns:

This function returns **STATUS_SUCCESS** (1) or **STATUS_FAIL** (0).

4.1.5.9 NFCP2P_NDEFURIRecordDecoder

Decode NDEF URI Records.

Prototype:

```
bool
NFCP2P_NDEFURIRecordDecoder(sNDEFURIRecord *sURIRecord,
                            uint8_t *pui8Buffer,
                            uint32_t ui32BufferLength)
```

Parameters:

sURIRecord is a pointer to the URIRecord structure into which to decode the data.

pui8Buffer is a pointer to the raw NFC data buffer to be decoded.

ui32BufferLength is the length of the raw NFC data buffer.

Description:

This function takes a raw NFC data buffer and decodes the data into a URI record data structure. It is assumed that the raw data buffer contains a URI record.

Returns:

This function returns **STATUS_SUCCESS** (1) or **STATUS_FAIL** (0).

4.1.5.10 NFCP2P_NDEFURIRecordEncoder

Encode NDEF URI Records.

Prototype:

```
bool
NFCP2P_NDEFURIRecordEncoder(sNDEFURIRecord sURIRecord,
                            uint8_t *pui8Buffer,
                            uint16_t ui16BufferMaxLength,
                            uint32_t *pui32BufferLength)
```

Parameters:

sURIRecord is the URI Record Structure to be encoded.

pui8Buffer is a pointer to the buffer to fill with the raw NFC data.

ui16BufferMaxLength is the maximum number of bytes the buffer can hold. This parameter is used to prevent writing past the end of the buffer.

pui32BufferLength is a pointer to the integer to hold the length of the raw NFC data buffer.

Description:

This function takes a URI Record structure and encodes it into a provided buffer in a raw NFC data format. The length of the data stored in the buffer is stored in **pui32BufferLength**.

Returns:

This function returns **STATUS_SUCCESS** (1) or **STATUS_FAIL** (0).

4.1.5.11 NFCP2P_proccessStateMachine

Processes low level stack.

Prototype:

```
tNFCP2PState
NFCP2P_proccessStateMachine(void)
```

Returns:

This function returns the current NFCP2P state.

The **tNFCP2PState** *return* parameter can be any of the following

- **NFC_P2P_PROTOCOL_ACTIVATION** - Polling/Listening for SENSF_REQ / SENSF_RES.
- **NFC_P2P_PARAMETER_SELECTION** - Setting the NFCIDs and bit rate
- **NFC_P2P_DATA_EXCHANGE_PROTOCOL** - Data exchange using the LLCP layer
- **NFC_P2P_DEACTIVATION** - Technology deactivation.

This function must be executed every 77 ms or less as defined by requirement 197 inside the Digital Protocol Technical Specification. When the g_eP2PMode is set to P2P_INITATIOR_MODE, this function sends a SENSF_REQ to check if there is a Target in the field, while blocking the main application. If there is no target in the field, it exits. When the g_eP2PMode is set to P2P_PASSIVE_TARGET_MODE, this function waits for command for 495 ms, while blocking the main application. If no commands are received or if any errors occurred, this function exits. Once a technology is activated for either P2P_INITATIOR_MODE or P2P_PASSIVE_TARGET_MODE, the main application can use g_eNFCP2PState when equal to NFC_P2P_DATA_EXCHANGE_PROTOCOL, to then call NFCP2P_sendPacket() to send data from the TRF7970A to a target/initiator. Furthermore when g_eNFCP2PState is NFC_P2P_DATA_EXCHANGE_PROTOCOL, the main application must check the receive state with the function NFCP2P_getReceiveState() each time NFCP2P_proccessStateMachine() is executed to ensure it handles the data as it is received.

Returns:

g_eNFCP2PState, which is the current P2P state.

4.1.5.12 tStatus NFCP2P_sendPacket (uint8_t $*$ *pui8DataPtr*, uint32_t *ui32DataLength*)

Sends a raw buffer of data to the SNEP stack to be transmitted.

Parameters:

pui8DataPtr is a pointer to the raw data to be sent.

ui32DataLength is the length of the raw data.

Description:

This function is used to send a data stream over NFC. The buffer resulting from a call to NFCP2P_NDEFMessageEncoder() should be fed to this function.

Returns:

Status of sent packet.

The **tStatus** *parameter* can be any of the following:

- **STATUS_FAIL** - The function exited with a failure.
- **STATUS_SUCCESS** - The function ended in succes.

5 NFCLib SNEP API

This section covers the SNEP API that can be used by the LLCP layer to send or receive data from a Peer-to-Peer device. For more information on SNEP, please see the LLCP document on the `NFC Forum website`.

5.1 NFC SNEP API Functions

Defines

- SNEP_MAX_BUFFER
- SNEP_MAX_PAYLOAD
- SNEP_VERSION

Enumerations

- tPacketStatus
- tSNEPCommands
- tSNEPConnectionStatus

Functions

- tSNEPConnectionStatus SNEP_getProtocolStatus (void)
- void SNEP_getReceiveStatus (tPacketStatus ∗peReceiveFlag, uint8_t ∗pui8length, uint8_-t ∗∗pui8DataPtr)
- void SNEP_init (void)
- void SNEP_processReceivedData (uint8_t ∗pui8RxBuffer, uint8_t ui8RxLength)
- uint8_t SNEP_sendRequest (uint8_t ∗pui8DataPtr, tSNEPCommands eRequestCmd)
- uint8_t SNEP_sendResponse (uint8_t ∗pui8DataPtr, tSNEPCommands eResponseCmd)
- void SNEP_setMaxPayload (uint8_t ui8MaxPayload)
- void SNEP_setProtocolStatus (tSNEPConnectionStatus eProtocolStatus)
- tStatus SNEP_setupPacket (uint8_t ∗pui8PacketPtr, uint32_t ui32PacketLength)

5.1.1 Detailed Description

Simple NDEF Exchange Protocol is an application protocol used by the LLCP layer to send / receive NDEFs between two NFC Forum Devices operating in Peer-to-Peer Mode (1 Target and 1 Initiator). For more information on SNEP, please read the NFC Simple NDEF Exchange Protocol Specification Version 1.0.

5.1.2 Define Documentation

5.1.2.1 SNEP_MAX_BUFFER

Definition:

```
#define SNEP_MAX_BUFFER
```

Description:

This is the maximum size of a fragment that is sent/received.

5.1.2.2 SNEP_MAX_PAYLOAD

Definition:

```
#define SNEP_MAX_PAYLOAD
```

Description:

Maximum size of the incoming payload.

5.1.2.3 SNEP_VERSION

Definition:

```
#define SNEP_VERSION
```

Description:

Simple NDEF protocol version specified in the specification.

5.1.3 Enumeration Documentation

5.1.3.1 tPacketStatus

Description:

RX packet status enumeration.

Enumerators:

RECEIVED_NO_FRAGMENT No pending received data.

RECEIVED_FIRST_FRAGMENT First fragment received from the client.

RECEIVED_N_FRAGMENT N fragment received from the client.

RECEIVED_FRAGMENT_COMPLETED Last fragment received from the client - packet completed.

5.1.3.2 tSNEPCommands

Description:

SNEPCommand request / responses enumeration.

Enumerators:

SNEP_REQUEST_CONTINUE See SNEP V1.0 Section 4.1.

SNEP_REQUEST_GET See SNEP V1.0 Section 4.2.
SNEP_REQUEST_PUT See SNEP V1.0 Section 4.3.
SNEP_REQUEST_REJECT See SNEP V1.0 Section 4.4.
SNEP_RESPONSE_CONTINUE See SNEP V1.0 Section 5.1.
SNEP_RESPONSE_SUCCESS See SNEP V1.0 Section 5.2.
SNEP_RESPONSE_NOT_FOUND See SNEP V1.0 Section 5.3.
SNEP_RESPONSE_EXCESS_DATA See SNEP V1.0 Section 5.4.
SNEP_RESPONSE_BAD_REQUEST See SNEP V1.0 Section 5.5.
SNEP_RESPONSE_NOT_IMPLEMENTED See SNEP V1.0 Section 5.6.
SNEP_RESPONSE_UNSUPPORTED_VER See SNEP V1.0 Section 5.7.
SNEP_RESPONSE_REJECT See SNEP V1.0 Section 5.8.

5.1.3.3 tSNEPConnectionStatus

Description:
SNEP Connection Status Enumeration.

Enumerators:
SNEP_CONNECTION_IDLE No ongoing transaction to/from the client.
SNEP_WRONG_VERSION_RECEIVED Wrong version received.
SNEP_CONNECTION_RECEIVED_FIRST_PACKET Received first fragment.
SNEP_CONNECTION_RECEIVING_N_FRAGMENTS Received n fragment.
SNEP_CONNECTION_WAITING_FOR_CONTINUE Waiting for continue response.
SNEP_CONNECTION_WAITING_FOR_SUCCESS Waiting for success response.
SNEP_CONNECTION_SENDING_N_FRAGMENTS Sending n fragment.
SNEP_CONNECTION_SEND_COMPLETE Send completed.
SNEP_CONNECTION_RECEIVE_COMPLETE Receive completed.
SNEP_CONNECTION_EXCESS_SIZE Received excess size request.

5.1.4 Function Documentation

5.1.4.1 SNEP_getProtocolStatus

Returns current SNEP Connection Status enumeration

Prototype:
```
tSNEPConnectionStatus
SNEP_getProtocolStatus(void)
```

Description:
This function returns the current SNEP status flag. It must be called inside LLCP_process-ReceivedData() to determine if further I-PDUs are required, which is when there are requests/responses queued.

Returns:
g_eSNEPConnectionStatus the current connection status flag.

5.1.4.2 SNEP_getReceiveStatus

Get RxStatus flag, Clear packet status flag,retrieve length of data and retrieve data

Prototype:

```
void
SNEP_getReceiveStatus( tPacketStatus *peReceiveFlag,
                       uint8_t *pui8length,
                       uint8_t **pui8DataPtr)
```

Parameters:

peReceiveFlag is a pointer to store the RX state status.
pui8length is a pointer to store the number of received bytes.
pui8DataPtr is a double pointer to store the pointer of data received.

Description:

The *peReceiveFlag* parameter can be any of the following:

- **RECEIVED_NO_FRAGMENT** - No Fragment has been received
- **RECEIVED_FIRST_FRAGMENT** - First fragment has been received.
- **RECEIVED_N_FRAGMENT** - N Fragment has been received.
- **RECEIVED_FRAGMENT_COMPLETED** - End of the fragment has been received.

This function must be called in the main application after the NFCP2P_proccessStateMachine() is called to ensure the data received is moved to another buffer and handled when a fragment is received.

Returns:

None

5.1.4.3 SNEP_init

Initialize the Simple NDEF Exchange Protocol driver.

Prototype:

```
void
SNEP_init(void)
```

Description:

This function must be called prior to any other function offered by the SNEP driver. This function initializes the SNEP status, Tx/Rx packet length and maximum payload size. This function must be called by the LLCP_init().

Returns:

None.

5.1.4.4 SNEP_processReceivedData

Processes the data received from a client/server.

Prototype:

```
void
SNEP_processReceivedData(uint8_t *pui8RxBuffer,
                         uint8_t ui8RxLength)
```

Parameters:

pui8RxBuffer is the starting pointer of the SNEP request/response received.

ui8RxLength is the length of the SNEP request/response received.

Description:

This function handles the requests/responses received inside an I-PDU in the LLCP layer. This function must be called inside LLCP_processReceivedData().

Returns:

None

5.1.4.5 SNEP_sendRequest

Sends request to the server.

Prototype:

```
uint8_t
SNEP_sendRequest(uint8_t *pui8DataPtr,
                 tSNEPCommands eRequestCmd)
```

Parameters:

pui8DataPtr is the start pointer where the request is written.

eRequestCmd is the request command to be sent.

Description:

The *eRequestCmd* parameter can be any of the following:

- **SNEP_REQUEST_CONTINUE** - Send remaining fragments
- **SNEP_REQUEST_GET** - Return an NDEF message
- **SNEP_REQUEST_PUT** - Accept an NDEF message
- **SNEP_REQUEST_REJECT** - Do not send remaining fragments

This function sends an SNEP request from the SNEP client to an SNEP server. It must be called from the LLCP_sendI() function.

Returns:

ui8offset, which is the length of the request written starting at **pui8DataPtr**.

5.1.4.6 SNEP_sendResponse

Sends response to the client.

Prototype:

```
uint8_t
SNEP_sendResponse(uint8_t *pui8DataPtr,
                  tSNEPCommands eResponseCmd)
```

Parameters:

pui8DataPtr is the start pointer where the response is written.

eResponseCmd is the response command to be sent.

Description:

The *eResponseCmd* parameter can be any of the following:

- **SNEP_RESPONSE_CONTINUE** - Continue send remaining fragments
- **SNEP_RESPONSE_SUCCESS** - Operation succeeded
- **SNEP_RESPONSE_NOT_FOUND** - Resource not found
- **SNEP_RESPONSE_EXCESS_DATA** - Resource exceeds data size limit
- **SNEP_RESPONSE_BAD_REQUEST** - Malformed request not understood
- **SNEP_RESPONSE_NOT_IMPLEMENTED** - Unsupported functionality requested
- **SNEP_RESPONSE_UNSUPPORTED_VER** - Unsupported protocol version
- **SNEP_RESPONSE_REJECT** - Do not send remaining fragments

This function sends an SNEP response from the SNEP server to an SNEP client. It must be called from the LLCP_sendI() function.

Returns:

ui8offset is the length of the response written starting at **pui8DataPtr**.

5.1.4.7 SNEP_setMaxPayload

Set the Maximum size of each fragment.

Prototype:

```
void
SNEP_setMaxPayload(uint8_t ui8MaxPayload)
```

Parameters:

ui8MaxPayload is the maximum size of each fragment.

Description:

This function must be called inside LLCP_processTLV() to define the maxium size of each fragment based on the Maximum Information Unit (MIU) supported by the target/initiator.

Returns:

None.

5.1.4.8 SNEP_setProtocolStatus

Sets current SNEP Connection Status enumeration

Prototype:

```
void
SNEP_setProtocolStatus( tSNEPConnectionStatus eProtocolStatus)
```

Parameters:

eProtocolStatus is the status flag used by the SNEP state machine SNEP_processReceived-Data() to send request/response. New sent transactions are allowed only when eProtocol-Status is set to **SNEP_CONNECTION_IDLE**.

Description:

The *eProtocolStatus* parameter can be any of the following:

- **SNEP_CONNECTION_IDLE** - No ongoing Tx/Rx
- **SNEP_WRONG_VERSION_RECEIVED** - Wrong Version Received
- **SNEP_CONNECTION_RECEIVED_FIRST_PACKET** - Received First Fragment
- **SNEP_CONNECTION_RECEIVING_N_FRAGMENTS** - Received N Fragment
- **SNEP_CONNECTION_WAITING_FOR_CONTINUE** - Waiting for Continue response
- **SNEP_CONNECTION_WAITING_FOR_SUCCESS** - Waiting for Success response
- **SNEP_CONNECTION_SENDING_N_FRAGMENTS** - Sending N Fragment
- **SNEP_CONNECTION_SEND_COMPLETE** - Send Completed
- **SNEP_CONNECTION_RECEIVE_COMPLETE** - Receive Completed
- **SNEP_CONNECTION_EXCESS_SIZE** - Received Excess Size request

This function is called inside LLCP_processReceivedData(), to set the *g_eSNEPConnectionStatus* flag to **SNEP_CONNECTION_IDLE** after a send transaction is completed to allow for further send transactions.

Returns:

None

5.1.4.9 SNEP_setupPacket

Set the global SNEP Packet Pointer and Length

Prototype:

```
tStatus
SNEP_setupPacket(uint8_t *pui8PacketPtr,
                 uint32_t ui32PacketLength)
```

Parameters:

pui8PacketPtr is the pointer to the first payload to be transmitted.

ui32PacketLength is the length of the total packet.

Description:

This function must be called by the main application to initialize the packet to be sent to the SNEP server.

Returns:

This function returns **STATUS_SUCCESS** (1) if the packet was queued, else it returns **STATUS_FAIL** (0).

6 nfclib LLCP API

This section covers the LLCP API that can be used by the DEP layer to establish and maintain a link between two Peer-to-Peer devices. For more information on LLCP, please see the LLCP document on the `NFC Forum website`.

6.1 NFC LLCP API Functions

Defines

- DSAP_SERVICE_DISCOVERY_PROTOCOL
- LLCP_MIU
- LLCP_MIUX_SIZE
- LLCP_SSAP_CONNECT_RECEIVED
- LLCP_SSAP_CONNECT_SEND

Enumerations

- tDisconnectModeReason
- tLLCPConnectionStatus
- tLLCPParamaeter
- tLLCPPduPtype
- tServiceName

Functions

- uint8_t LLCP_addTLV (tLLCPParamaeter eLLCPparam, uint8_t ∗pui8TLVBufferPtr)
- uint16_t LLCP_getLinkTimeOut (void)
- void LLCP_init (void)
- tStatus LLCP_processReceivedData (uint8_t ∗pui8RxBuffer, uint8_t ui8PduLength)
- void LLCP_processTLV (uint8_t ∗pui8TLVBufferPtr)
- uint8_t LLCP_sendCC (uint8_t ∗pui8PduBufferPtr)
- uint8_t LLCP_sendCONNECT (uint8_t ∗pui8PduBufferPtr)
- uint8_t LLCP_sendDISC (uint8_t ∗pui8PduBufferPtr)
- uint8_t LLCP_sendDM (uint8_t ∗pui8PduBufferPtr, tDisconnectModeReason eDmReason)
- uint8_t LLCP_sendI (uint8_t ∗pui8PduBufferPtr)
- uint8_t LLCP_sendRR (uint8_t ∗pui8PduBufferPtr)
- uint8_t LLCP_sendSYMM (uint8_t ∗pui8PduBufferPtr)
- tStatus LLCP_setNextPDU (tLLCPPduPtype eNextPdu)
- uint8_t LLCP_stateMachine (uint8_t ∗pui8PduBufferPtr)

6.1.1 Detailed Description

Logical Link Control Protocol is the NFC transport layer used to open and close a virtual link used to transfer NDEFs between two devices in peer-to-peer mode via the Simple NDEF Exchange Protocol. For more information on LLCP, please read the Logical Link Control Protocol Specification Version 1.1.

6.1.2 Define Documentation

6.1.2.1 DSAP_SERVICE_DISCOVERY_PROTOCOL

Definition:

```
#define DSAP_SERVICE_DISCOVERY_PROTOCOL
```

Description:

Destination Service Access Point for discovery.

6.1.2.2 LLCP_MIU

Definition:

```
#define LLCP_MIU
```

Description:

The LLCP_MIU is the maximum information unit supported by the LLCP layer. This information unit may be included in each LLCP packet depending on the PDU type. The minimum must be 128.

6.1.2.3 LLCP_MIUX_SIZE

Definition:

```
#define LLCP_MIUX_SIZE
```

Description:

The LLCP_MIUX_SIZE is the value for the LLCP_MIUX TLV used in LLCP_addTLV().

6.1.2.4 LLCP_SSAP_CONNECT_RECEIVED

Definition:

```
#define LLCP_SSAP_CONNECT_RECEIVED
```

Description:

Source Service Access Point when receiving.

6.1.2.5 LLCP_SSAP_CONNECT_SEND

Definition:

```
#define LLCP_SSAP_CONNECT_SEND
```

Description:

Source Service Access Point when sending.

6.1.3 Enumeration Documentation

6.1.3.1 tDisconnectModeReason

Description:

Disconnected Mode Reasons Enumerations.

Enumerators:

DM_REASON_LLCP_RECEIVED_DISC_PDU See LLCP Section 4.3.8.

DM_REASON_LLCP_RECEIVED_CONNECTION_ORIENTED_PDU See LLCP Section 4.3.8.

DM_REASON_LLCP_RECEIVED_CONNECT_PDU_NO_SERVICE See LLCP Section 4.3.8.

DM_REASON_LLCP_PROCESSED_CONNECT_PDU_REQ_REJECTED See LLCP Section 4.3.8.

DM_REASON_LLCP_PERMNANTLY_NOT_ACCEPT_CONNECT_WITH_SAME_SSAP See LLCP Section 4.3.8.

DM_REASON_LLCP_PERMNANTLY_NOT_ACCEPT_CONNECT_WITH_ANY_SSAP See LLCP Section 4.3.8.

DM_REASON_LLCP_TEMMPORARILY_NOT_ACCEPT_PDU_WITH_SAME_SSSAPT See LLCP Section 4.3.8.

DM_REASON_LLCP_TEMMPORARILY_NOT_ACCEPT_PDU_WITH_ANY_SSSAPT See LLCP Section 4.3.8.

6.1.3.2 tLLCPConnectionStatus

Description:

LLCP Connection Status Enumeration.

Enumerators:

LLCP_CONNECTION_IDLE No Tx/Rx ongoing.

LLCP_CONNECTION_ESTABLISHED When a virtual link is created either when we send a CONNECT PDU and receive a CC PDU, or when we receive a CONNECT PDU and respond a CC PDU.

LLCP_CONNECTION_SENDING When sending data via SNEP.

LLCP_CONNECTION_RECEIVING When receiving data via SNEP.

6.1.3.3 tLLCPParamaeter

Description:
LLCP Parameter Enumerations.

Enumerators:

LLCP_VERSION See LLCP V1.1 Section 4.5.1.
LLCP_MIUX See LLCP V1.1 Section 4.5.2.
LLCP_WKS See LLCP V1.1 Section 4.5.3.
LLCP_LTO See LLCP V1.1 Section 4.5.4.
LLCP_RW See LLCP V1.1 Section 4.5.5.
LLCP_SN See LLCP V1.1 Section 4.5.6.
LLCP_OPT See LLCP V1.1 Section 4.5.7.
LLCP_SDREQ See LLCP V1.1 Section 4.5.8.
LLCP_SDRES See LLCP V1.1 Section 4.5.9.

6.1.3.4 tLLCPPduPtype

Description:
PDU Type Enumerations.

Enumerators:

LLCP_SYMM_PDU See LLCP V1.1 Section 4.3.1.
LLCP_PAX_PDU See LLCP V1.1 Section 4.3.2.
LLCP_AGF_PDU See LLCP V1.1 Section 4.3.3.
LLCP_UI_PDU See LLCP V1.1 Section 4.3.4.
LLCP_CONNECT_PDU See LLCP V1.1 Section 4.3.5.
LLCP_DISC_PDU See LLCP V1.1 Section 4.3.6.
LLCP_CC_PDU See LLCP V1.1 Section 4.3.7.
LLCP_DM_PDU See LLCP V1.1 Section 4.3.8.
LLCP_FRMR_PDU See LLCP V1.1 Section 4.3.9.
LLCP_SNL_PDU See LLCP V1.1 Section 4.3.10.
LLCP_I_PDU See LLCP V1.1 Section 4.3.11.
LLCP_RR_PDU See LLCP V1.1 Section 4.3.12.
LLCP_RNR_PDU See LLCP V1.1 Section 4.3.13.
LLCP_RESERVED_PDU See LLCP V1.1 Section 4.3.14.

6.1.3.5 tServiceName

Description:
Service Name Enumerations - Only support SNEP_SERVICE.

6.1.4 Function Documentation

6.1.4.1 uint8_t LLCP_addTLV (tLLCPParamaeter *eLLCPparam*, uint8_t $*$ *pui8TLVBufferPtr*)

Adds a LLCP parameter to the LLCP PDU with the Type Length Value (TLV) format.

Parameters:

eLLCPparam is the LLCP type that will be added.
pui8TLVBufferPtr is the pointer where the TLV is written

The **eLLCPparam** *parameter* can be any of the following:

- **LLCP_VERSION** - Version Number
- **LLCP_MIUX** - Maximum Information Unit Extension
- **LLCP_WKS** - Well-Known Service List
- **LLCP_LTO** - Link Timeout
- **LLCP_RW** - Receive Window Size
- **LLCP_SN** - Service Name
- **LLCP_OPT** - Option
- **LLCP_SDREQ** - Service Discovery Request
- **LLCP_SDRES** - Service Discovery Response
- **LLCP_ERROR** - Reserved (used ro return length of 0)

This function is used to add a LLCP Parameter to the LLCP PDU to include more information about the LLCP layer. This function must be called inside LLCP_sendCONNECT(), LLCP_sendCC(), NFCDEP_sendATR_REQ() and NFCDEP_sendATR_RES().

Returns:

ui8PacketLength Length of the LLCP Parameter added to the LLCP.

6.1.4.2 uint16_t LLCP_getLinkTimeOut (void)

Gets link timeout

This function returns the Link Timeout, which may be modified if the LLCP_processTLV() function processes a LLCP_LTO TLV.

Returns:

g_ui16LLCPlto *the* link timeout.

6.1.4.3 void LLCP_init (void)

Initializes the Logical Link Control Protocol layer.

This function must be called prior to any other function offer by the LLCP driver. This function initializes the acknowledge packets, the current service enabled, and the next PDU for the LLCP_-stateMachine(), and also initializes the SNEP layer with SNEP_init().

Returns:
None

6.1.4.4 tStatus LLCP_processReceivedData (uint8_t ∗ *pui8RxBuffer*, uint8_t *ui8PduLength*)

Processes LLCP Data Received.

Parameters:
pui8RxBuffer is the start pointer of the LLCP data received.
ui8PduLength is the length of the LLCP PDU received.

This function is used to handle the LLCP portion of the DEP_REQ / DEP_RES PDU. This function must be called inside NFCDEP_processReceivedRequest() and NFCDEP_processReceivedData().It currently does not support to handle the following PDUs : LLCP_PAX_PDU, LLCP_AGF_PDU, LLCP_UI_PDU, LLCP_FRMR_PDU, LLCP_SNL_PDU, LLCP_RNR_PDU, and LLCP_RESERVED_PDU.

Returns:
eLLCPStatus is the boolean status if the command was processed (1) or not (0).

6.1.4.5 void LLCP_processTLV (uint8_t ∗ *pui8TLVBufferPtr*)

Processes the LLCP Parameter TLV.

Parameters:
pui8TLVBufferPtr is the pointer to the Type value of the TLV.

This function processes the LLCP Parameters included in the ATR_RES. This function must be called inside the NFCDEP_processReceivedData(), to initialize the g_ui8LLCPmiu and g_ui16LLCPlto if they are included as part of ATR_RES.

Returns:
None

6.1.4.6 uint8_t LLCP_sendCC (uint8_t ∗ *pui8PduBufferPtr*)

Send CC message

Parameters:
pui8PduBufferPtr is the start pointer to store the CC PDU.

This function adds a CC PDU starting at pui8PduBufferPtr. For more details on this PDU, read LLCP V1.1 Section 4.3.7.

Returns:
ui8IndexTemp is the length of the CC PDU.

6.1.4.7 uint8_t LLCP_sendCONNECT (uint8_t ∗ *pui8PduBufferPtr*)

Send CONNECT message

Parameters:

pui8PduBufferPtr is the start pointer to store the CONNECT PDU.

This function adds a CONNECT PDU starting at pui8PduBufferPtr.For more details on this PDU read LLCP V1.1 Section 4.3.5.

Returns:

ui8IndexTemp is the length of the CONNECT PDU.

6.1.4.8 uint8_t LLCP_sendDISC (uint8_t ∗ *pui8PduBufferPtr*)

Send DISC message

Parameters:

pui8PduBufferPtr is the start pointer to store the DISC PDU.

This function adds a DISC PDU starting at pui8PduBufferPtr.For more details on this PDU read LLCP V1.1 Section 4.3.6.

Returns:

ui8IndexTemp is the length of the DISC PDU.

6.1.4.9 uint8_t LLCP_sendDM (uint8_t ∗ *pui8PduBufferPtr*, tDisconnectModeReason *eDmReason*)

Send DM message

Parameters:

pui8PduBufferPtr is the start pointer to store the DM PDU.

eDmReason is the enumeration of the disconnection reason.

The *eDmReason* parameter can be any of the following:

- **DM_REASON_LLCP_RECEIVED_DISC_PDU**
- **DM_REASON_LLCP_RECEIVED_CONNECTION_ORIENTED_PDU**
- **DM_REASON_LLCP_RECEIVED_CONNECT_PDU_NO_SERVICE**
- **DM_REASON_LLCP_PROCESSED_CONNECT_PDU_REQ_REJECTED**
- **DM_REASON_LLCP_PERMNANTLY_NOT_ACCEPT_CONNECT_WITH_SAME_SSAP**
- **DM_REASON_LLCP_PERMNANTLY_NOT_ACCEPT_CONNECT_WITH_ANY_SSAP**
- **DM_REASON_LLCP_TEMMPORARILY_NOT_ACCEPT_PDU_WITH_SAME_SSSAPT**
- **DM_REASON_LLCP_TEMMPORARILY_NOT_ACCEPT_PDU_WITH_ANY_SSSAPT**

This function adds a DM PDU starting at pui8PduBufferPtr with a dm_reason. For more details on this PDU read LLCP V1.1 Section 4.3.8.

Returns:
ui8IndexTemp is the length of the DM PDU.

6.1.4.10 uint8_t LLCP_sendI (uint8_t ∗ *pui8PduBufferPtr*)

Send I message

Parameters:
pui8PduBufferPtr is the start pointer to store the I PDU.

This function adds a I PDU starting at pui8PduBufferPtr.For more details on this PDU read LLCP V1.1 Section 4.3.10.

Returns:
ui8IndexTemp is the length of the I PDU.

6.1.4.11 uint8_t LLCP_sendRR (uint8_t ∗ *pui8PduBufferPtr*)

Send RR message

Parameters:
pui8PduBufferPtr is the start pointer to store the RR PDU.

This function adds a RR PDU starting at pui8PduBufferPtr.For more details on this PDU read LLCP V1.1 Section 4.3.11.

Returns:
ui8IndexTemp is the length of the RR PDU.

6.1.4.12 uint8_t LLCP_sendSYMM (uint8_t ∗ *pui8PduBufferPtr*)

Send SYMM message

Parameters:
pui8PduBufferPtr is the start pointer to store the SYMM PDU.

This function adds a SYMM PDU starting at pui8PduBufferPtr.For more details on this PDU read LLCP V1.1 Section 4.3.1.

Returns:
ui8IndexTemp is the length of the SYMM PDU.

6.1.4.13 tStatus LLCP_setNextPDU (tLLCPPduPtype *eNextPdu*)

Set next PDU, return SUCCESS or FAIL

Parameters:
eNextPdu is the LLCP PDU to set next.

The *eNextPdu* parameter can be any of the following:

- **LLCP_SYMM_PDU** - See LLCP standard document section 4.3.1
- **LLCP_PAX_PDU** - See LLCP standard document section 4.3.2
- **LLCP_AGF_PDU** - See LLCP standard document section 4.3.3
- **LLCP_UI_PDU** - See LLCP standard document section 4.3.4
- **LLCP_CONNECT_PDU** - See LLCP standard document section 4.3.5
- **LLCP_DISC_PDU** - See LLCP standard document section 4.3.6
- **LLCP_CC_PDU** - See LLCP standard document section 4.3.7
- **LLCP_DM_PDU** - See LLCP standard document section 4.3.8
- **LLCP_FRMR_PDU** - See LLCP standard document section 4.3.9
- **LLCP_SNL_PDU** - See LLCP standard document section 4.3.10
- **LLCP_I_PDU** - See LLCP standard document section 4.3.11
- **LLCP_RR_PDU** - See LLCP standard document section 4.3.12
- **LLCP_RNR_PDU** - See LLCP standard document section 4.3.13
- **LLCP_RESERVED_PDU** - See LLCP standard document section 4.3.14
- **LLCP_ERROR_PDU** - Unknown PDU

This function is used to modify the next LLCP PDU. For example when we need to set the next PDU to be LLCP_CONNECT_PDU, to initiate a transfer. For more information please see the LLCP document from the NFC Forum.

Returns:
eSetNextPduStatus SUCCESS if g_eNextPduQueue was modified, else return FAIL

6.1.4.14 uint8_t LLCP_stateMachine (uint8_t ∗ *pui8PduBufferPtr*)

Prepares the LLCP packet to be transmitted.

Parameters:
pui8PduBufferPtr is the start pointer to add the LLCP PDU.

This function is used to add the LLCP portion of the DEP_REQ / DEP_RES PDU. This function must be called inside NFCDEP_sendDEP_REQ() and NFCDEP_sendDEP_RES(). It currently does not support sending the following PDUs : LLCP_PAX_PDU, LLCP_AGF_PDU, LLCP_UI_PDU, LLCP_-FRMR_PDU, LLCP_SNL_PDU, LLCP_RNR_PDU, and LLCP_RESERVED_PDU.

Returns:
ui8PacketLength is the length of the LLCP PDU added to the pui8PduBufferPtr.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		