MC45
NFC PROGRAMMING GUIDE

MC45
NFC PROGRAMMING GUIDE

MN000145A02

Rev. A

February 2015

Zebra Technologies Corporation
Lincolnshire, IL U.S.A
http://www.zebra.com

Revision History

Changes to the original guide are listed below:

Change	Date	Description
Rev A	10/2013	Initial release
Rev A	2/2015	Zebra rebranding

TABLE OF CONTENTS

ABOUT THIS GUIDE

Introduction

This guide describes the Zebra Application Programmer's Interface (API) which provides a way for application developers to programmatically access the NFC stack functionality and aids in new application development.

The main features of NFC software stack are:

- Supports NFC functionality such as NFC card reader and card emulation.
- Supports identification, reading and writing to the following tags:
  - NFC tag types 1-4
  - Mifare
  - Calypso
  - ISO 15693 NXP i-CODE
  - Felica
  - TI TagIT.

Chapter Descriptions

Topics covered in this guide are as follows:

- *Chapter 1, Functions* provides a detailed overview of all of the functions available for MC45 NFC programming including a description, function prototype, parameters, return values, comments and an example for each function.
- *Chapter 2, Callback Prototypes* provides information on all of the available callback functions for MC45 NFC programming including a description, function prototype, parameters, and return values for each function.
- *Chapter 3, Structures* describes the available structures for MC45 NFC programming including a description and structure definition for each.
- *Chapter 4, Return Values* provides a table of all possible return values along with a description of each.
- *Chapter 5, Programming Overview* provides an overview of NFC programming giving step-by-step procedures and examples on how to initialize and deinitialize the NFC stack, view the NFC capabilities of the

system, NFC configuration, NFC radio and polling services, configuring application types and modes and creating/reading NDEF messages.

Notational Conventions

The following conventions are used in this document:

- *Italics* are used to highlight the following:
  - Chapters and sections in this and related documents
  - Dialog box, window and screen names
  - Drop-down list and list box names
  - Check box and radio button names
  - Icons on a screen.
- **Bold** text is used to highlight the following:
  - Key names on a keypad
  - Button names on a screen or window.
- bullets (•) indicate:
  - Action items
  - Lists of alternatives
  - Lists of required steps that are not necessarily sequential
- Sequential lists (e.g., those that describe step-by-step procedures) appear as numbered lists.

✓ ***NOTE*** This symbol indicates something of special interest or importance to the reader. Failure to read the note will not result in physical harm to the reader, equipment or data.

⚠ ***CAUTION*** This symbol indicates that if this information is ignored, the possibility of data or material damage may occur.

⚠ ***WARNING!*** **This symbol indicates that if this information is ignored the possibility that serious personal injury may occur.**

Service Information

If you have a problem with your equipment, contact Zebra Global Customer Support for your region. Contact information is available at: http://www.zebra.com/support.

When contacting Zebra Global Customer Support, please have the following information available:

- Serial number of the unit
- Model number or product name
- Software type and version number.

Zebra responds to calls by E-mail, telephone or fax within the time limits set forth in support agreements.

If your problem cannot be solved by Zebra Global Customer Support, you may need to return your equipment for servicing and will be given specific directions. Zebra is not responsible for any damages incurred during shipment if the approved shipping container is not used. Shipping the units improperly can possibly void the warranty.

If you purchased your business product from a Zebra business partner, contact that business partner for support.

CHAPTER 1 FUNCTIONS

Introduction

This chapter provides a detailed overview of all of the functions available for MC45 NFC programming including a description, function prototype, parameters, return values, comments and examples.

NFC_Init()

Description

Initializes the NFC stack. This needs to be called before any other NFC APIs.

Function Prototype

NFC_STATUS NFC_Init (void)

Parameters

None

Return Values

NFC_SUCCESS	NFC_SUCCESS in case of success.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources like memory.
NFC_ERROR_DLL_NOT_AVAILABLE	NFC Stack DLL has not been loaded.
NFC_ERROR_NOT_SUPPORTED	NFC is Not Supported on the terminal.
NFC_FAILED	If any other error occurs.

Comments

Individual firmware version, stack version, etc. can be obtained through NFC_GetCapabilities.

See Also

NFC_DeInit()

Example

```
/* Initialize the NFC stack */
if(nStatus = NFC_Init() != NFC_SUCCESS)
{
   printf("Error %d: Cannot initialize the NFC stack.\n", nStatus);
   exit();
}
```

NFC_DeInit()

Description

De-initializes the NFC stack. Any open handles will be closed by the NFC stack.

Function Prototype

NFC_STATUS NFC_DeInit(void)

Parameters

None

Return Values

NFC_SUCCESS	NFC_SUCCESS in case of success.
NFC_FAILED	If any error occurs.

Comments

None

See Also

NFC_Init()

Example

```
/* De-Initialize the NFC stack */
if(nStatus = NFC_DeInit() != NFC_SUCCESS)
{
   printf("Error %d: DeInit NFC stack.\n", nStatus);
   return -1;
}
```

NFC_GetVersion()

Description

This function obtains the version information of the NFC stack.

Function Prototype

NFC_STATUS NFC_GetVersion (NFC_Version *pNfcVersion)

Parameters

[out] pNFCVersion	A pointer of NFC_Version passed to the function. The function returns the consolidated version number of the NFC Stack.

Return Values

NFC_SUCCESS	Indicates success
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Ex. memory)
NFC_FAILED	If any error occurs

Comments

Individual firmware version, stack version, etc. can be obtained through NFC_GetCapabilities.

See Also

NFC_GetCapabilities

Example

```
/* Initialize the NFC stack */
if(nStatus = NFC_GetVersion(&nfcVersion) != NFC_SUCCESS)
{
   printf("Error %d: Failed to get NFC Version.\n", nStatus);
   exit();
}
```

NFC_GetCapabilities()

Description

This function returns the capabilities of the stack, the version number of the firmware, the vendor name, etc. The capabilities are stored in a structure.

Function Prototype

NFC_STATUS NFC_GetCapabilities (NFC_CapStruct *pCapabilities)

Parameters

[out] pCapabilties	A pointer to a structure for receiving the capability values.

Return Values

NFC_SUCCESS	Indicates success.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example: memory).
NFC_FAILED	If any error occurs.

Comments

None

See Also

None

Example

```
NFC_CapStruct capValues;
NFC_STATUS status;
status = NFC_GetCapabilities(&capValues);
...
```

NFC_GetConfig()

Description

This function returns the current configuration values of the stack.

Function Prototype

NFC_STATUS NFC_GetConfig (NFC_Config *psConfigValues)

Parameters

[out] psConfigValues	The configuration values.

Return Values

NFC_SUCCESS	Indicates success.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example: memory).
NFC_FAILED	If any error occurs.

Comments

None

See Also

NFC_SetConfig()

Example

```
NFC_Config configValues;
NFC_STATUS status;
status = NFC_GetConfig(&configValues);
...
```

NFC_SetConfig()

Description

This function sets configuration values for the stack. See *NFC_Version on page 3-2* for the values that can be configured.

Function Prototype

NFC_STATUS NFC_SetConfig (NFC_Config sConfigValues)

Parameters

[in] psConfigValues	The configuration values.

Return Values

NFC_SUCCESS	The operation is performed successfully.
NFC_ERROR_BAD_PARAMETER	A wrong value is detected for a parameter.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example: memory).
NFC_FAILED	If any other error occurs.

Comments

Important: Before modifying the configuration, you must first obtain the system configuration with properly populated fields using NFC_GetConfig(), and then continue to modify the required fields. Subsequently, set the configuration using NFC_SetConfig().

See Also

NFC_GetConfig(), NFC_SetDefaultConfig(), NFC_RegisterForConfigChange()

Example

```
NFC_Config configValues;
NFC_STATUS status;

If (NFC_GetConfig(&configValues) == SUCCESS)
{
   configValues.PollingTimeout = 60;
   configValues.passkey = {'1','1','1','1','\0'};
   If (NFC_SetConfig(configValues) != SUCCESS)
   {
...
```

NFC_SetDefaultConfig()

Description

This function sets the default configuration values for the stack. The default configuration values are defined in *NFC_Version on page 3-2*.

Function Prototype

NFC_STATUS NFC_SetDefaultConfig (Void)

Parameters

None

Return Values

NFC_SUCCESS	The operation is performed successfully.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example: memory).
NFC_FAILED	If any other error occurs.

Comments

None

See Also

NFC_SetConfig()

Example

```
NFC_STATUS status;
status = NFC_SetDefaultConfig();
...
```

NFC_RegisterForConfigChange()

Description

This function registers a callback that needs to be invoked whenever a configuration is changed by another application. This is usually used to indicate if the other application changed configuration.

Function Prototype

NFC_STATUS NFC_RegisterForConfigChange (NFC_Config_Callback *pHandler,PVOID pPassbackParameter, NFC_HANDLE*phConfigChangeHandle)

Parameters

[in] pHandler	A pointer on the handler function invoked when any configuration changes.
[in] pPassbackParameter	A parameter transmitted to the handler function.
[out] phConfigChangeHandle	A pointer on a variable valued with the handle of the registration. This value may be used with NFC_CloseHandle() to unregister the handler function.

Return Values

NFC_SUCCESS	The operation is performed successfully.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example: memory).
NFC_FAILED	If any other error occurs.

Comments

None

See Also

None

Example

```
NFC_Handle configChangeHandle;

NFC_STATUS status;

status = NFC_RegisterForConfigChange(configChangeHandler, NULL,
configChangeHandle);

...
```

NFC_CloseHandle()

Description

Frees the resources allocated for a handle. The handle becomes invalid after the call to NFC_CloseHandle(). Using this handle subsequently will cause an error.

Function Prototype

NFC_STATUS NFC_CloseHandle (NFC_HANDLE hHandle)

Parameters

[in] hHandle	The handle of the connection to close.

Return Values

NFC_SUCCESS	NFC_SUCCESS in case of success.
NFC_ERROR_BAD_HANDLE	A bad handle value is detected.
NFC_FAILED	Indicates if any other errors.

Comments

The following handles may be used with NFC_CloseHandle():

- The event registry handles. Closing an event registry handle causes the un-registration of the event handler function.
- The connection handles.
- The NFC message handles.

See Also

NFC_CloseAllHandles()

Example

```
NFC_HANDLE aTagEventRegistry;
/* Register TAG handler */
...
/* Un-Register TAG handler */
NFC_CloseHandle(aTagEventRegistry);
```

NFC_CloseAllHandles()

Description

Frees the resources allocated by all handles in the application. The handles become invalid after the call to NFC_CloseAllHandles(). Using those handles subsequently will cause an error.

Function Prototype

NFC_STATUS NFC_CloseAllHandles (void)

Parameters

None

Return Values

NFC_SUCCESS	NFC_SUCCESS in case of success.
NFC_FAILED	Indicates if any other errors.

Comments

None

See Also

NFC_CloseHandle()

Example

```
// Close all Handles
NFC_CloseAllHandles();
```

NFC_RegisterForTagDetection()

Description

This function registers a handler function to listen to tag detection. After a successful registration, the handler function is called each time a tag is detected. NFC_CloseHandle() should be called to unregister the function.

If TagsToBeDetected is set to null, every tag supported by the stack is detected.

Function Prototype

NFC_STATUS NFC_RegisterForTagDetection (NFC_ TagDetection _Callback *pHandler, PVOID pPassbackParameter, const NFC_Tags TagsToBeDetected, NFC_HANDLE *phTagDetectionHandle)

Parameters

[in] pHandler	A pointer on the handler function called to listen to the tag detection.
[in] pPassbackParameter	A parameter transmitted to the handler function.
[in] TagsToBeDetected	A structure with TRUE values corresponding to type of tags to be detected.
[out] phTagDetectionHandle	A pointer on a variable valued with the handle of the registration. This value may be used with NFC_CloseHandle() to unregister the handler function.

Return Values

NFC_SUCCESS	NFC_SUCCESS in case of success.
NFC_ERROR_BAD_PARAMETER	A bad parameter value.
NFC_ERROR_FEATURE_NOT_SUPPORTED	The requested feature is not supported.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources. (Example: memory).
NFC_FAILED	Indicates if any other errors.

Comments

Calling this function turns on the NFC radio if it is off and starts the polling process for all registered technologies.

See Also

NFC_GetCapabilities()

Example

```
/* Register tag event handler */
NFC_Tags TagsToBeDetected;
TagsToBeDetected.MiFareUL = TRUE;
TagsToBeDetected.MiFareStd = TRUE;
TagsToBeDetected.ISO14443A = TRUE;
```

```
NFC_STATUS nStatus = NFC_RegisterForTagDetection(static_TagDetectionHandler,
NULL, TagsToBeDetected, &hTagDetectionHandle);
/* Get the result of the tag detection */
void static_TagDetectionHandler (void *pPassbackParameter, NFC_HANDLE
hConnection,
NFC_TagInfo TagInfo, NFC_STATUS nStatus)
{
   /* Perform tag operations based on detected tag*/
      If (TagInfo.TagsAndProtocols.MiFareUL == TRUE)
      {
      }
      else if (TagInfo.TagsAndProtocols.MiFareStd == TRUE)
      {
      }
   …
}
```

NFC_RegisterForMessage()

Description

Retrieves NDEF messages from NFC tags. The function registers a callback function to be called each time a matching message is read from a tag. Value 'zero' for nTNF, registers for all NDEF messages. A valid value in nTNF and NULL for pTypesArray, registers for all types of the registered TNF format.

The un-registration is done by closing the handle returned in the NDEFMessageRegistrationHandle parameter using NFC_CloseHandle().

Function Prototype

NFC_STATUS NFC_RegisterForMessage (NFC_Message _Callback* pHandler, PVOID pPassbackParameter, WORD nTNF, const TCHAR* pTypesArray[], WORD nNoOfTypes, NFC_HANDLE* NDEFMessageRegistrationHandle)

Parameters

[in] pHandler	A pointer on the handler function.
[in] pPassbackParameter	A parameter transmitted to the handler function.
[in] nTNF	The message TNF value. An nTNF value of 0 indicates registration for all NDEF messages. The valid values for nTNF are: TNF-WKT (Well-Known Type), TNF-MIME, TNF-AbsURI, TNF-EXT (External Type) and TNF-UNK (Unknown Type).
[in] pTypesArray	An array of types to filter for. A value of NULL indicates registration for all types with TNF value indicated by nTNF.
[in] nNoOfTypes	No. of types to filter for; the types are in pTypesArray.
[out] NDEFMessageRegistrationHandle	A pointer on a variable valued with the handle of the registration. This value may be used with NFC_CloseHandle() to unregister the handler function.

Return Values

NFC_SUCCESS	The operation was performed successfully.
NFC_ERROR_BAD_PARAMETER	A bad parameter value.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example: memory).
NFC_FAILED	If any other error occurred.

Comments

NFC_RegisterForMessage registers for reading message present on any type of tag (Type 1, 2, 3, 4). The TNF and Type of the first record in the message is considered for comparing with the registered TNF and Types.

Calling this function turns on the NFC radio if it is off and starts the polling process for all technologies.

Several handlers can be registered at the same time, allowing the registration of separate handlers for the same or different messages types. If a received message matches several registered handlers, all the matching handlers are called in sequence.

Each handler should call the function NFC_CloseHandle() to allow further call to the other listeners.

See Also

NFC_RegisterForTagDetection()

Example

```
/* Register NDEF Message handler */
NFC_Handle hMessageDetectionHandle;
Char NDEFType_Sp[] = "Sp";
Char *TypesArray[1] = NDEFType_Sp;
{
   …
NFC_STATUS nStatus = NFC_RegisterForMessage(static_NDEFMessageHandler, NULL,
TNF_WKT, TypesArray, 1, &hMessageDetectionHandle);
   ….
}
/* Get the result of the message detection */
void static_NDEFMessageHandler (void* pPassbackParameter,
NFC_HANDLE hConnection, NFC_HANDLE hMessageHandle, WORD nTNF, WORD nNoOfRecords,
DWORD nPayloadLengths[], WORD nTypeLengths[], WORD nIDLengths[],
NFC_STATUS nStatus)
{
/* Get the values of the No. of Records, Payload lengths, Type lengths, etc,*/
   ….
}
```

NFC_GetRecord()

Description

This function retrieves information about a record.

Function Prototype

NFC_STATUS NFC_GetRecord (NFC_HANDLE hMessageHandle, WORD nRecordNo, WORD *tnf, PCHAR *pBuffer, PCHAR *pTypeString, PCHAR *pID)

Parameters

[in] hMessageHandle	The message handle.
[in] nRecordNo	The record no. in the message which has to be read.
[out] tnf	TNF value of the record. Valid values for nTNF are: TNF-WKT (Well-Known Type), TNF-MIME, TNF-AbsURI, TNF-EXT (External Type) and TNF-UNK (Unknown Type).
[out] pBuffer	Payload of the record.
[out] pTypeString	Type Value of the record.
[out] pID	ID value in the record.

Return Values

NFC_SUCCESS	The operation was performed successfully.
NFC_ERROR_BAD_HANDLE	A bad handle was detected.
NFC_ERROR_INCORRECT_RECORD_NO	Incorrect record no. provided.
NFC_ERROR_BAD_NDEF_FORMAT	Incorrect NDEF format.
NFC_ERROR_NDEF_UNKNOWN	The NDEF message is unknown.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Ex.memory).
NFC_FAILED	If any other error had occurred.

Comments

If NULLs are passed for pTypeString, pBuffer and PID in the structure, those values are not returned.

See Also

None

Example

```
NFC_HANDLE hMessageHandle;
WORD tnf;
CHAR Type[40], ID[40], Buffer[100];
/* To retrieve record */
```

```
if (NFC_GetRecord(hMessageHandle, nRecordNo, &tnf, Buffer, Type, ID) !=
NFC_SUCCESS)
{
   printf("Error: NFC_GetRecord failed !\n");
   break;
}
...
/* Once done with Message handle */
NFC_CloseHandle (hMessageHandle);
```

NFC_GetEnclosedMessageInfo()

Description

Extracts message information enclosed in the payload of a record. Pointers to arrays of record handles, payload lengths, types and IDs are filled with the information of the enclosed message.

Function Prototype

NFC_STATUS NFC_GetEnclosedMessageInfo (NFC_HANDLE hMessageHandle, WORD nRecordNo, NFC_HANDLE *phEnclMessageHandle, WORD *noOfRecords, WORD *pnTNF, DWORD PayloadLength[], WORD TypeLength[], WORD IDLength[])

Parameters

[in] hMessageHandle	Message handle of the record containing enclosed message.
[in] nRecordNo	Record number of the message containing enclosed message.
[out] phEnclMessageHandle	Message handle to the enclosed message.
[out] noOfRecord	Number of records in the enclosed message.
[out] pnTNF	TNF value of the first record.
[out] PayloadLength	Lengths of payload in each record of the enclosed message.
[out] TypeLength	Lengths of type in each record of the enclosed message.
[out] IDLength	Lengths of ID in each record of the enclosed message.

Return Values

NFC_SUCCESS	The operation was performed successfully.
NFC_ERROR_BAD_HANDLE	A bad handle was detected.
NFC_ERROR_INCORRECT_RECORD_NO	Incorrect record no. provided.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example: memory).
NFC_ERROR_BAD_NDEF_FORMAT	Incorrect NDEF format.
NFC_ERROR_NDEF_UNKNOWN	Unknown NDEF message format.
NFC_FAILED	If any other error had occurred.

Comments

If NULLs are passed instead of pointers, those values are not returned.

Initially NULL can be sent for the arrays and number of records obtained. Subsequently, the function can be called with array pointers.

The user has the values of TNF and TYPE of the record to determine if it has an enclosed message. E.g.: TNF value of TNF-WKT and TYPE value of “Sp” for a record would let the user know as per the NFC-Forum specification, that the record has an enclosed message.

See Also

None

Example

```
NFC_HANDLE hMessageHandle, enclMessageHandle;
WORD noOfRecords, nTNF;
/* Get info of the message enclosed in a record */
if (NFC_GetEnclosedMessageInfo ( hMessageHandle, nRecordNo, &enclMessageHandle,
&noOfRecords, &nTNF, NULL, NULL, NULL) == NFC_SUCCESS)
{
   // Allocate arrays for required number of Records
   if (NFC_GetEnclosedMessageInfo ((hMessageHandle, nRecordNo,
   &enclMessageHandle, &noOfRecords, &nTNF, PayloadLengths[], TypeLengths[],
   IDLengths[]) != NFC_SUCCESS)
   {
      // ERROR
   }
...
```

NFC_GetMessageContent()

Description

The function returns the binary buffer containing a message.

Function Prototype

NFC_STATUS NFC_GetMessageContent (NFC_HANDLE hMessage, PCHAR pMessageBuffer, DWORD *pnActualBufferLength)

Parameters

[in] hMessage	The message handle.
[out] pMessageBuffer	A pointer on a buffer receiving the message content.
[out] pnActualBufferLength	Length of the message.

Return Values

NFC_SUCCESS	The operation was performed successfully.
NFC_ERROR_BAD_HANDLE	A bad handle value.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example: memory).
NFC_FAILED	If any other error had occurred.

Comments

A call to this function with NULL for pMessageBuffer can be made to obtain the actual buffer length. Memory can be allocated, and a call made to get the message content.

See Also

NFC_GetRecord()

Example

```
NFC_HANDLE hMessage;
DWORD nMesageLength;
if(NFC_GetMessageContent(hMessage, NULL, &nMessageLength)== NFC_SUCCESS)
{
   //Allocate memory for the buffer
   if(NFC_GetMessageContent(hMessage, pMessage,  &nMessageLength) != NFC_SUCCESS)
   {
      printf( "Error: NFC_GetMessageContent\n" );
      return;
   }
}
Else
```

```
{
   printf( "Error: NFC_GetMessageContent\n" );
   return;
}
...
/* Once done with message handle */
NFC_CloseHandle (hMessage);
```

NFC_CreateMessage()

Description

Creates a new NDEF message. The message created is a blank message. Call NFC_AddRecord() to add records into the message.

Function Prototype

NFC_STATUS NFC_CreateMessage (NFC_HANDLE* phMessage)

Parameters

[out] phMessage	A pointer on a variable valued with the handle of the new message.

Return Values

NFC_SUCCESS	The operation was performed successfully.
NFC_ERROR_OUT_OF_RESOURCE	Out of memory.
NFC_FAILED	If any other error had occurred.

Comments

None

See Also

None

Example

```
NFC_HANDLE hMessage;
if(NFC_CreateMessage(&hMessage) != NFC_SUCCESS)
{
   printf("Error: Cannot create the new message.\n");
}
```

NFC_AddRecord()

Description

This function adds a record to the message. This record is added to the tail end of the existing records in the message.

Function Prototype

NFC_STATUS NFC_AddRecord (NFC_Record sRecord, NFC_HANDLE hMessageHandle)

Parameters

[in] sRecord	Record structure containing information to be added as a record to the message.
[in] hMessageHandle	Handle of the message to which the record has to be added.

Return Values

NFC_SUCCESS	The operation was performed successfully.
NFC_ERROR_BAD_HANDLE	A bad handle value provided.
NFC_ERROR_BAD_NDEF_FORMAT	Incorrect NDEF formatted record.
NFC_ERROR_OUT_OF_RESOURCE	Out of memory.
NFC_FAILED	If any other error occurred.

Comments

None

See Also

None

Example

```
NFC_HANDLE hMessageHandle;
NFC_Record NDEFRecord;
CHAR URI[] = {'m', 's', 'i', '.', 'c', 'o', 'm', '/', 'm', 'c', '4', '5', '\0'};
DWORD nBufferLength;

NDEFRecord.Type = TYPE_URI;
NDEFRecord.URIRecord.URIType = HTTP_WWW;
NDEFRecord.URIRecord.Payload = URI;

/* Creates a Message */
if(NFC_CreateMessage(&hMessageHandle) == NFC_SUCCESS)
{
        /* Creates a record using the buffer */
        if(NFC_AddRecord(NDEFRecord, hMessageHandle) != NFC_SUCCESS)
        {
            printf("Error: Cannot Add Record.\n");
        }
        // Add further records
    }
    else
    {
        printf("Error: Cannot create the new message.\n");

        ...

    /* Done with the record handle */
    NFC_CloseHandle( hMessageHandle );
```

NFC_WriteMessage()

Description

This function writes a message to the current tag.

Function Prototype

NFC_STATUS NFC_WriteMessage (NFC_HANDLE hConnection, NFC_HANDLE *phMessage[], WORD nNoOfMessages)

Parameters

[in] hConnection	The connection handle.
[in] phMessage	An array of message handles.
[in] nNoOfMessages	Number of messages to be written.

Return Values

NFC_SUCCESS	The operation was performed successfully.
NFC_ ERROR_BAD_HANDLE	Bad handle provided.
NFC_ ERROR_TAG_NOT_NDEF	Tag is not NDEF compliant/format.
NFC_ ERROR_TAG_DISCONNECTED	Tag is no longer in proximity.
NFC_ ERROR_TAG_LOCKED	Tag is locked for writing.
NFC_ERROR_TAG_WRITE	Any other tag Write error.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example, memory).
NFC_FAILED	If any other error had occurred.

Comments

Multiple message writing capability of the stack can be checked using NFC_GetCapabilities.

✓ ***NOTE*** Currently, only one NDEF message can be written.

See Also

None

Example

```
NFC_HANDLE hConnection;
NFC_HANDLE hMessage[1];

// Create Message and Add records
// Register for Tag
```

```
/* Write the message on TAG */
NFC_WriteMessage( hConnection, &hMessage, 1);
...

/* All operations done with tag */
NFC_CloseHandle (hConnection);
NFC_CloseHandle (hMessage);
```

NFC_ExchangeData()

Description

This function exchanges raw data with the object accessible with this connection.

Function Prototype

NFC_STATUS NFC_ExchangeData (NFC_HANDLE hConnection, const PCHAR pReaderToTagBuffer, DWORD nReaderToTagBufferLength, PCHAR pTagToReaderBuffer, DWORD *nTagToReaderBufferMaxLength)

Parameters

[in] hConnection	The connection handle.
[in] pReaderToTagBuffer	A pointer on the buffer containing the data to send to the tag.
[in] nReaderToTagBufferLength	The length in bytes of the data to send to the tag.
[out] pTagToReaderBuffer	A pointer on the buffer receiving the data returned by the tag.
[in/out] pnTagToReaderBufferMaxLength	The maximum length in bytes of the buffer. pTagToReaderBuffer as input, returns the actual read data.

Return Values

NFC_SUCCESS	The operation was performed successfully.
NFC_ ERROR_BAD_HANDLE	Bad handle provided.
NFC_ ERROR_COMMAND_NOT_SUPPORTED	Command not supported response from tag.
NFC_ ERROR_INVALID_PARAMETER	Invalid parameter response from tag.
NFC_ ERROR_TAG_DISCONNECTED	Tag is no longer in proximity.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example: memory).
NFC_FAILED	If any other error had occurred.

Comments

None

See Also

None

Example

```
char pReaderToTagBuffer[] = {0x30, 0x04, 0x26, 0xee};
char pTagToReaderBuffer[8];
DWORD readLength = 8;

status = NFC_ExchangeData(hConnection, pReaderToTagBuffer, 4, pTagToReaderBuffer,
&readLength)
```

NFC_NDEFFormat()

Description

This function formats the tag as per NDEF.

CAUTION This process is irreversible. Once a tag has been made 'read only', it cannot be used to write again.

Function Prototype

NFC_STATUS NFC_NDEFFormat (PCHAR *pScrtKey, WORD ScrtKeyLength, NFC_HANDLE hConnection)

Parameters

[in] pScrtKey	Buffer containing the Secret key.
[in] ScrtKeyLength	Length of the secret key.
[in] hConnection	Handle to the connection.

Return Values

NFC_SUCCESS	The operation has started successfully.
NFC_ ERROR_BAD_HANDLE	Bad handle provided.
NFC_ ERROR_TAG_ALREADY_NDEF	Tag is already NDEF compliant/format.
NFC_ ERROR_TAG_DISCONNECTED	Tag is no longer in proximity.
NFC_ERROR_INVALID_PARAMETER	Invalid parameter response from tag.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources like memory.
NFC_FAILED	If any other error occurred.

Comments

None

See Also

None

Example

```
Char scrtKey[10];
WORD scrtKeyLength,
NFC_HANDLE hConnection;
NFC_STATUS status;


……
// fill in scrtKey and scrtKeyLength
```

```
status = NFC_NDEFFormat(scrtKey, scrtKeyLength, hConnection);
...
```

NFC_MakeReadOnly()

Description

This function makes the tag read-only.

CAUTION This process is irreversible. Once a tag has been made 'read only', it cannot be used to write again.

Function Prototype

NFC_STATUS NFC_MakeReadOnly (PCHAR *pScrtKey, WORD ScrtKeyLength, NFC_HANDLE hConnection)

Parameters

[in] pScrtKey	Buffer containing the secret key.
[in] ScrtKeyLength	Length of the secret key.
[in] hConnection	Handle to the connection.

Return Values

NFC_SUCCESS	The operation has started successfully.
NFC_ ERROR_BAD_HANDLE	Bad handle provided.
NFC_ ERROR_TAG_ALREADY_READONLY	Tag is already 'read only'.
NFC_ ERROR_TAG_DISCONNECTED	Tag is no longer in proximity.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources (Example: memory).
NFC_ERROR_INVALID_PARAMETER	Invalid parameter response from tag.
NFC_FAILED	If any other error had occurred.

Comments

None

See Also

None

Example

```
Char scrtKey[10];
WORD scrtKeyLength;
NFC_HANDLE hConnection;
NFC_STATUS status;
……
// fill in scrtKey and scrtKeyLength
```

```
status = NFC_MakeReadOnly(scrtKey, scrtKeyLength, hConnection);
...
```

NFC Card Emulation APIs

NFC_RegisterForHostCardEmulation()

Description

This function registers a card for emulation in the stack. Initially a blank card is setup in the stack and a message handle is returned to the application. The application can add records to this message. When the application has added all the records it wants, it can call NFC_EnableCE().

Function Prototype

NFC_STATUS NFC_RegisterForHostCardEmulation (NFC_Message _Callback* pHandler, PVOID pPassbackParameter, DWORD cardSize, BOOL readAccess, BOOL writeAccess, NFC_HANDLE* phRegistrationHandle, NFC_HANDLE* phMessageHandle)

Parameters

[in] pHandler	A pointer on the handler function invoked, when an external reader has made an update to the card.
[in] pPassbackParameter	A parameter transmitted to the handler function.
[in] cardSize	Size of the card to be emulated. The stack will allocate memory for this size.
[in] readAccess	True will provide read acesss to external readers. False will not provide read access.
[in] writeAccess	True will provide write acesss to external readers. False will not provide write access.*
[out] phRegistrationHandle	A pointer on a variable valued with the handle of the registration. This value may be used with NFC_CloseHandle() to unregister the handler function.
[out] phMessageHandle	A pointer for the message handle.

Return Values

NFC_SUCCESS	NFC_SUCCESS in case of success.
NFC_ERROR_BAD_PARAMETER	A bad parameter value.
NFC_ERROR_FEATURE_NOT_SUPPORTED	The requested feature is not supported.
NFC_ERROR_OUT_OF_RESOURCE	Out of resources like memory.
NFC_FAILED	If any other error occurs.

Comments

*Currently write access for the emulated host card is not supported. Parameters write_accesss, pHandler and pPassbackParameter are available in the API as a place holder for future support.

See Also

NFC_GetCapabilities()

Example

```
{
   nStatus = NFC_RegisterForHostCardEmulation(
   NULL, //Not expecting callback, since write access is False
   NULL, 512, True, False, &cardEmulRegistryHandle, &messageHandle);

   If (nStatus == NFC_SUCCESS)
   {
      /*  Add record and enable CE here */
      NFC_Record NDEFRecord;
      CHAR URI[] = {'m', 's', 'i', '.', 'c', 'o', 'm', '/', 'm', 'c', '4', '5',
      '\0'};
      DWORD nBufferLength;
      NDEFRecord.Type = TYPE_URI;
      NDEFRecord.URIRecord.URIType = HTTP_WWW;
      NDEFRecord.URIRecord.Payload = URI;
      if(NFC_AddRecord(NDEFRecord, messageHandle) == NFC_SUCCESS)}
      {
         NFC_EnableCE(&ceHandle, CECallback, NULL, CE_HOST);
      }
   }
}
```

NFC_EnableCE()

Description

Enables card emulation; the source of card emulation can be selected through CE_Source parameter. The source can be UICC, embedded secure element or host. NFC_GetCapabilities can be used to verify the supported sources.

Function Prototype

NFC_STATUS NFC_EnableCE (NFC_HANDLE* CEHandle, NFC_CE_Callback *pCallback, PVOID pPassbackParameter, WORD nCE_Source)

Parameters

[out] CEHandle	Handle to the registration.
[in] pCallback	A pointer on the callback function.
[in] pPassbackParameter	A parameter transmitted to the callback function.
[in] nCE_Source	The source for CE.

Return Values

NFC_SUCCESS	The operation is performed successfully.
NFC_ERROR_OUT_OF_RESOURCE	Out of memory.
NFC_ERROR_FEATURE_NOT_SUPORTED	CE/ CE source not supported.
NFC_FAILED	If any other error occurred.

Comments

None

See Also

NFC_DisableCE(), NFC_GetCapabilities()

Example

```
NFC_Handle ceHandle;
NFC_STATUS status;
status = NFC_EnableCE(&ceHandle, CECallback, NULL, CE_UICC);
...
```

NFC_DisableCE()

Description

This function disables the card emulation.

Function Prototype

NFC_STATUS NFC_DisableCE (NFC_HANDLE CEHandle)

Parameters

[in] CEHandle	Handle to the Registration.

Return Values

NFC_SUCCESS	The operation is performed successfully.
NFC_ERROR_BAD_HANDLE	Bad handle provided.
NFC_FAILED	If any other error occurred.

Comments

None

See Also

NFC_EnableCE()

Example

```
NFC_Handle ceHandle;
NFC_STATUS status;
status = NFC_DisableCE(ceHandle);
...
```

NFC Bluetooth Pairing APIs

NFC_EnableBTPairing()

Description

This function enables Bluetooth Pairing through NFC.

Function Prototype

NFC_STATUS NFC_EnableBTPairing (NFC_HANDLE* BTHandle, NFC_BT_Callback *pCallback, PVOID pPassbackParameter)

Parameters

[out] BTHandle	Handle to the registration.
[in] pCallback	A pointer on the callback function.
[in] pPassbackParameter	A parameter transmitted to the callback function.

Return Values

NFC_SUCCESS	The operation was performed successfully.
NFC_ERROR_OUT_OF_RESOURCE	Out of memory.
NFC_ERROR_FEATURE_NOT_SUPPORTED	Feature not supported.
NFC_FAILED	If any other error had occurred.

Comments

NFC radio is turned on, if not already on and polling process started for all types of tags/technologies.

The callback is invoked whenever a Bluetooth device has been paired through NFC.

See Also

NFC_DisableBTPairing()

Example

```
NFC_Handle BTHandle;
NFC_STATUS status;
status = NFC_EnableBTPairing(&BTHandle, BTCallback, NULL);
...
```

NFC_DisableBTPairing()

Description

This function disables the Bluetooth Pairing through NFC.

Function Prototype

NFC_STATUS NFC_DisableBTPairing (NFC_HANDLE BTHandle)

Parameters

[in] BTHandle	Handle to the Registration.

Return Values

NFC_SUCCESS	The operation was performed successfully.
NFC_ERROR_BAD_HANDLE	Bad handle provided.
NFC_FAILED	If any other error had occurred.

Comments

None

See Also

NFC_EnableBTPairing()

Example

```
NFC_Handle BTHandle;
NFC_STATUS status;
status = NFC_DisableBTPairing(BTHandle);
...
```

CHAPTER 2 CALLBACK PROTOTYPES

Introduction

This chapter provides information on all of the available callback functions for MC45 NFC programming including a description, function prototype, parameters, and return values.

NFC_Config_Callback

Description

This is the type of the callback function to be implemented to receive changed configuration data.

Prototype

typedef void NFC_Config_Callback (PVOID pPassbackParameter, NFC_Config ConfigValues, NFC_STATUS nResult)

Parameters

[in] pPassbackParameter	The parameter given to the function initiating the operation.
[in] ConfigValues	Structure containing the configuration values.
[in] nResult	The result code of the operation. NFC_SUCCESS in case of success. The other error codes are described in the documentation of the calling function.

Return Values

Not applicable.

NFC_CE_Callback

Description

This is the type of the callback function to receive notification of CE events.

Function Prototype

typedef void NFC_CE_Callback (PVOID pPassbackParameter, WORD nEventCode, NFC_STATUS nResult)

Parameters

[in] pPassbackParameter	The parameter given to the function registering the event handler.
[in] nEventCode	The event code.
[in] nResult	Result code of the operation: NFC_SUCCESS in case of success. The other error codes are described in the documentation of the calling function.

Return Values

Not applicable.

NFC_BT_Callback

Description

This is the type of the callback function to receive notification of Bluetooth pairing.

Function Prototype

typedef void NFC_BT_Callback (PVOID pPassbackParameter, NFC_BTPairingInfo PairedDevice, NFC_STATUS nResult)

Parameters

[in] pPassbackParameter	The parameter given to the function registering the event handler.
[in] PairedDevice	The Bluetooth paired device information.
[in] nResult	The result code of the operation: NFC_SUCCESS in case of success: The other error codes are described in the documentation of the calling function.

Return Values

Not applicable.

NFC_TagDetection_Callback

Description

This callback function is needed to be provided through NFC_RegisterForTagDetection(). The handler function is called each time a tag of the specified type is detected.

Function Prototype

typedef void NFC_TagDetection_callback (PVOID pPassbackParameter, NFC_HANDLE hConnection, NFC_TagInfo TagInfo, NFC_STATUS nResult)

Parameters

[in] pPassbackParameter	The value provided to the function NFC_RegisterForDetection()) when the registration was performed.
[in] hConnection	The handle of the new connection with the tag.
[in] TagInfo	A structure containing the information of the tag.
[in] nResult	NFC_SUCCESS in case of success. Another error code in case of error during the registration.

Return Values

Not applicable.

NFC_Message_Callback

Description

This is the type of the handler function to be implemented to handle the detected NDEF messages.

This callback function is needed to be provided by NFC_RegisterForMessage() or NFC_RegisterForHostCardEmulation()*. The handler function is called each time a tag with the specified NDEF message type is detected or an external reader has updated the emulated card containing an NDEF message.

Function Prototype

typedef void NFC_Message_Callback (PVOID pPassbackParameter, NFC_HANDLE hConnection, NFC_HANDLE hMessageHandle, WORD nTNF, WORD nNoOfRecords, DWORD nPayloadLengths[], WORD nTypeLengths[], WORD nIDLengths[], NFC_STATUS nResult)

Parameters

[in] pPassbackParameter	The value provided to the function NFC_RegisterForMessage() or NFC_RegisterForHostCardEmulation() when the registration was performed.
[in] hConnection	The handle of the new connection with the tag, in case of callback from Card Emulation, this will be NULL.
[in] hMessageHandle	Contains message handle of the message.
[in] nTNF	TNF value of the first record in the message.
[in] nNoOfRecords	No. of records in the message.
[in] nPayloadLengths[]	Contains the lengths of payloads in an array.
[in] nTypeLengths[]	Contains the lengths of type of each record in an array.
[in] nIDLengths[]	Contains the length of ID of each record in an array.
[in] nResult	NFC_SUCCESS in case of success: Other error code in case of error during the registration.

Return Values

Not applicable.

Comments

* Currently host card emulation - write access is not supported, therefore callbacks due to update of emulated card would not be invoked.

CHAPTER 3 STRUCTURES

Introduction

This chapter describes the available structures for the MC45 NFC programming including a description and structure definition.

NFC_HANDLE

Description

The type of the handles.

Structure Definition

```
typedef uint32 NFC_HANDLE
```

STRUCT_INFO

Description

The STRUCT_INFO structure enables extensibility of API structures. A STRUCT_INFO structure is embedded in each structure of the API to describe the memory allocated and used by that structure. If the structure grows in future revisions of the API, the STRUCT_INFO information can be used to determine which fields are valid for use. Refer to Zebra EMDK v2.0 for details.

Structure Definition

```
typedef struct tagSTRUCT_INFO
{
   DWORD dwAllocated;
   DWORD dwUsed;
} STRUCT_INFO;
```

NFC_Version

Description

This structure provides the major, minor and revision information of the API DLL.

Structure Definition

```
typedef  struct NFC_version
{
   STRUCT_INFO structInfo;
   WORD MajorVersion;
   WORD MinorVersion;
   WORD Revision;
} NFC_Version;
```

NFC_Config

Description

This structure has fields of various configurable parameters.

Structure Definition

```
typedef  struct tagNFC_Config
{
   STRUCT_INFO structInfo;
   uint16 PollingTimeout; // in sec, 0 - Timeout disabled, default : 90
   char passkey[6]; // Max of 6 characters, First Character '/0' is no
   Passkey(Default is No Passkey)
} NFC_Config;
```

NFC_Record

Description

This structure has fields for constructing various records. The record for URI and text can be created using types TYPE_URI and TYPE_TEXT in the type field. All other record types can be created using TYPE_GENERAL in the type field. Data for each type of the record can be filled with the corresponding union structure. Refer to the programmer's guide for examples.

Structure Definition

```
typedef  struct tagNFC_Record
{
   STRUCT_INFO structInfo;
   WORD Type;
   union
   {
      struct
      {
         WORD TNF;
         PCHAR TYPE;
         PCHAR ID;
         WORD IDLength;
         PCHAR Payload;
         DWORD PayloadLength;
      } General;
      struct
      {
         WORD CharEncoding;
         PCHAR Language;
         PCHAR Payload;
         DWORD PayloadLength;
      } Text;
      struct
      {
         WORD Type;
         PCHAR Payload;
         DWORD PayloadLength;
      } URI;
} NFC_Record;
```

NFC_Tags

Description

This structure has fields of various tags. This structure is in APIs and also part of other structures. The meaning of the individual fields set to true or false depends upon the context.

Structure Definition

```
typedef struct tagNFC_Tags
{
   STRUCT_INFO structInfo;
   BOOL MiFareUL;
   BOOL MiFareStd;
   BOOL MifareULC;
   BOOL ISO14443_4A;
   BOOL ISO14443_4B;
   BOOL ISO15693;
   BOOL FeliCa;
   BOOL Jewel;
   BOOL Desfire;
   BOOL MiFarePlus;
   BOOL NXPIcode;
   BOOL Topaz;
   BOOL Calypso;
   BOOL TITagIt;
} NFC_Tags;
```

NFC_CapStruct

Description

This structure has further structures in its fields for conveying different capabilities.

- **ModesSupported:** The various modes supported by the stack are indicated through this structure. The field set to true indicates the mode is supported, false indicates the mode is not supported.
- **TagsSupported:** The various tags/technologies supported are indicated through this structure. A field set to true indicates the tag/technology is supported, false indicates the tag/technology is not supported.
- **TagsFormattingSupported:** The support for formatting various tags is indicated through this structure. A field set to true indicates formatting that tag is supported, false indicates formatting the tag is not supported.
- **TagsMakeReadOnlySupported:** The support for making various tags "read only" is indicated through this structure. A field set to true indicates Making those types of tag 'read only' is supported, false indicates making read only for those types of Tags is not supported.
- **DeviceInformation:** Specific details of the NFC chip used like - firmware version no., vendor name, etc.

Structure Definition

```
typedef Struct tagNFC_CapStruct {
   STRUCT_INFO structInfo;
   Struct
   {
      BOOL NFC_P2P;
      BOOL NFC_Reader_Writer;
      BOOL NFC_CE;
      BOOL NFC_CE_UICC;
      BOOL NFC_CE_SE;
      BOOL NFC_CE_Host;
      BOOL Multiple_NDEFMessages;
      BOOL NFC_LLCP;
      BOOL NFC_SNEP;
      BOOL NFC_NPP;
      BOOL NFC_BT_StaticHandover;
      BOOL NFC_BT_DynamicHandover;
      ...
   } ModesSupported;
   NFC_versionStackVersion;
   NFC_Tags  TagsSupported;
   NFC_Tags  TagsFormattingSupported;
   NFC_Tags TagsMakeReadOnlySupported;
   enum
   {
      PN544C3 = 01;
```

```
    } Device;

    union
    {
       struct
       {
          uint32 hal_version;
          uint32 fw_version
          uint32 hw_version
          uint8 model_id;
          uint8 hci_version;
          utf8 vendor_name[VENDOR_NAME_LEN];
          uint8 full_version[NXP_FULL_VERSION_LEN];
          Nfc_sSupProtocol ReaderSupProtocol;
          /* < Supported protocols (Bitmapped) in Reader mode. */
          Nfc_sSupProtocol EmulationSupProtocol;
          /*< Supported protocols   (Bitmapped) in Emulation   mode. */
          char firmware_update_info;
       } sPN544C3;

    } sDeviceCapabilities;
} NFC_CapStruct;

Where

typedef struct tagNfc_sSupProtocol
{
   unsigned int MifareUL    : 1;
   unsigned int MifareStd   : 1;
   unsigned int ISO14443_4A : 1;
   unsigned int ISO14443_4B : 1;
   unsigned int ISO15693    : 1;
   unsigned int Felica      : 1;
   unsigned int NFC         : 1;
   unsigned int Jewel       : 1;
} Nfc_sSupProtocol;
```

NFC_TagInfo

Description

Not applicable.

Structure Definition

```
typedef struct tagNFC_TagInfo
{
   STRUCT_INFO structInfo;
   DWORD nTagType, // The tag type.
   DWORD nSerialNumberLength, // The length in bytes of the serial number of tag.
   BOOL bIsNDEF, // Indicating NDEF formatted.
   BOOL bIsLocked // Flag indicating if the tag is locked for write
   operation.
   DWORD nTagSize // The number of bytes available to writes messages.
   DWORD nFreeSpaceSize // The size of the remaining free space in the tag. The
   free space is the number of bytes available to writes messages.
   DWORD sSerialNumber // The serial number of a tag.
} NFC_TagInfo;
```

NFC_BTPairingInfo

Description

Not applicable.

Structure Definition

```
typedef struct tagNFC_BTPairingInfo
{
   // BT Address
   STRUCT_INFO structInfo;
   WORD aBTAddress [BT_PAIRING_SIZE_OF_BT_ADDRESS]
   // Device Class property existing Indicator
   BOOL bHasDeviceClass
   // Device Class value
   WORD aBTDeviceClass [BT_PAIRING_SIZE_OF_DEVICE_CLASS]
   // Simple Pairing hash property existing Indicator
   BOOL bHasSimplePairingHash
   // Simple Pairing   //hash value
   WORD aBTSimplePairingHash [BT_PAIRING_SIZE_OF_PAIRING_HASH]
   // Simple Pairing Randomizer property existing Indicator
   BOOL bHasSimplePairingRandomizer
   // Simple Pairing Randomizer value
   WORD aBTSimplePairingRandomizer [BT_PAIRING_SIZE_OF_PAIRING_RANDOM]
   // Local Name Length value
   WORD nLocalNameLength
   // Complete value Indicator
   BOOL bIsCompleteLocalName
   // Local Name value.
   WORD aBTLocalName [BT_PAIRING_MAX_LOCAL_NAME_LENGTH]
   // Flag Length value
   WORD nFlagLength
   // Flags values
   WORD aBTFlag [BT_PAIRING_MAX_FLAGS_LENGTH]
   // UUID Length value
   WORD nUUIDListLength
   // Class number indicator
   WORD nUUIDServiceClassNumber
   // Complete Value indicator
   BOOL bIsCompleteUUIDList

   // UUID List Value
   WORD aBTUUIDList [BT_PAIRING_MAX_UUID_LIST_LENGTH]
   // Manufacturer data length value
   WORD nManufacturerDataLength
   // Byte array containing the manufacturer data
   WORD aBTManufacturerData [BT_PAIRING_MAX_MANUFACTURER_DATA_LENGTH]
} NFC_BTPairingInfo;

Where
// Bluetooth address length
#define BT_PAIRING_SIZE_OF_BT_ADDRESS 6
// Device Class length.
#define BT_PAIRING_SIZE_OF_DEVICE_CLASS 3
// Pairing hash size
#define BT_PAIRING_SIZE_OF_PAIRING_HASH 16
// Pairing random size
#define BT_PAIRING_SIZE_OF_PAIRING_RANDOM 16
// Max local name length
#define BT_PAIRING_MAX_LOCAL_NAME_LENGTH 128
// Max UUID List length (in byte)
#define BT_PAIRING_MAX_UUID_LIST_LENGTH 128
// Max flag length
#define BT_PAIRING_MAX_FLAGS_LENGTH 4
// Max manufacturer data length
#define BT_PAIRING_MAX_MANUFACTURER_DATA_LENGTH 128
```

CHAPTER 4 RETURN VALUES

Introduction

This chapter provides a list of all possible return values along with a description.

NFC_STATUS

NFC_SUCCESS	The function was completed successfully and no error occurred.
NFC_ERROR_BAD_HANDLE	A bad handle value was detected.
NFC_ERROR_BAD_PARAMETER	A wrong value was detected for a parameter.
NFC_ERROR_OUT_OF_RESOURCE	Resources error.
NFC_ERROR_BAD_TAG_FORMAT	Incorrect tag format.
NFC_ERROR_BAD_NDEF_FORMAT	Incorrect NDEF format.
NFC_ERROR_TIMEOUT	The communication failed because a timeout expired.
NFC_ERROR_FEATURE_NOT_SUPPORTED	The requested feature is not supported.
NFC_ERROR_NDEF_UNKNOWN	The NDEF message type is unknown.
NFC_ERROR_LOCKED_TAG	A write operation failed because the tag is locked or protected against writing operation.
NFC_ERROR_TAG_WRITE	An error is detected in the data written into a tag.
NFC_ERROR_INCORRECT_RECORD_NO	Incorrect record number provided as parameter.
NFC_ERROR_UICC_COMMUNICATION	Error in communicating with UICC SE.
NFC_ERROR_TAG_DISCONNECTED	Tag is no longer in Proximity.
NFC_ERROR_TAG_NOT_NDEF	Tag is not NDEF Formatted.
NFC_ERROR_TAG_ALREADY_NDEF	Tag is already NDEF Formatted.

NFC_ERROR_TAG_ALREADY_READONLY	Tag is already Made Read only and locked for Writing.
NFC_ERROR_WRONG_RTD	Incorrect RTD type.
NFC_ERROR_TAG_WRITE	Error in the tag write operation.
NFC_ERROR_COMMAND_NOT_SUPPORTED	Command Not supported response from tag.
NFC_ERROR_OPERATION_PENDING	Last operation not yet completed.
NFC_ERROR_INVALID_PARAMETER	Invalid Parameters response from tag.
NFC_ERROR_DLL_NOT_AVAILABLE	DLL not loaded.
NFC_ERROR_NOT_SUPPORTED	NFC not supported in the terminal.
NFC_ERROR_INCORRECT_RECORD_NO	Record No. provided is invalid.
NFC_FAILED	Other failures.
NFC_ERROR_BUFFER_TOO_LARGE	The buffer to be written to a Tag is large than the Tag capacity.
NFC_BATTERY_CRITICAL	Battery percentage is below 6% and polling can’t be started.
NFC_STATUS_FLIGHT_MODE_ON	Flight mode is ON and polling can’t be started.
NFC_NOTIFY_TARGET_LOST	Target moved out of the proximity.
NFC_STATUS_UICC_IN_PROGRESS	UICC card emulation is in progress.
NFC_STATUS_HOH_IN_PROGRESS	Host card emulation is in progress.
NFC_STATUS_RW_IN_PROGRESS	Reader mode is in progress.
NFC_CE_CLOSED	Card emulation stopped due to reader mode enabled.
NFC_STACK_ALREADY_INITIALISED	Stack is already initialized.
NFC_STACK_NOT_INITIALISED	Stack is not initialized.

CHAPTER 5 PROGRAMMING OVERVIEW

Introduction

This chapter provides an overview of NFC programming providing step-by-step procedures and examples on how to initialize and de-initialize the NFC stack, view the NFC capabilities of the system, NFC configuration, NFC radio and polling services, configuring application types and modes and creating/reading NDEF messages.

NFC Components

The NFC API is provided as a single DLL named *NFCAPI32.dll.* It provides its interface via 'C' function calls. The API is provided to the developer as a set of files that include the following:

• nfcapi32.dll	Dynamic Link Library containing 'C' NFC functions.
• nfcapi32.lib	Link file used by applications when linking in NFC functions.
• nfcapi.h	Include files that define NFC API functions.
• nfcdefs.h	NFC API defines and structures.

NFC Programming Overview

This Programming Guide is intended to assist programmers with the creation of NFC applications for MC45 NFC mobile computers running a Microsoft Windows Mobile operating system. It provides all of the details necessary for calling functions that access NFC functionality. The EMDK installer contains header file **`NfcApi.h`** with the function declaration and return error from the NFC stack which needs to be handled by the application. NFC CPL gets loaded at the boot time itself and then loads the **`NfcApi.dll`**. The **`nfcmanager.dll`** is a driver which also gets loaded at the boot time.

The NFC stack is based on client-driver architecture. Within client-driver architecture, multiple client applications simultaneously access the NFC Zebra APIs.

NFC Stack Initialization & De-Initialization

The first function called by the NFC client application should be NFC_Init(). This function initializes the NFC stack context.

When the NFC functionality is no longer required, the client should call NFC_DeInit() to release the resources allocated by NFC for this client.

If multiple applications exist, each of them need to initialize the stack separately by calling NFC_Init() function. The NFC device and stack initialization needs to be performed by an application irrespective of another application having already done this (i.e. if you have two applications always called in sequence, even though the first application initializes NFC, the other application also has to initialize NFC).

NFC Capabilities of the System

You can obtain the NFC capabilities of the system by using the API NFC_GetCapabilities(). Using the information provided, the application can decide whether the features it requires are available in the system.

The API NFC_GetCapabilities() will return following set of information:

1. **Device info:** hardware chip used, firmware version and hardware version.
2. **NFC Modes supported:** This indicates whether NFC-P2P, NFC-READER/WRITER and NFC-CARD EMULATION are supported.
3. If NFC-P2P mode is supported, the protocols and capabilities.
4. If NFC-READER/WRITER mode is supported. The following capabilities on the various tags would be available:
   a. Tag types supported.
   b. NDEF reading/writing capability for each tag type.
   c. NDEF formatting capability for each tag type.
   d. Make read only capability for each tag type.
5. If NFC-CARDEMULATION is supported, the types of card emulation supported are: UICC, embedded secure element and Host.

Configuring NFC

The APIs related to configuration are:

1. The API for getting default (Current) parameters is NFC_GetConfig().

2. The API for setting configured parameter is NFC_SetConfig().
3. The API for registering for any configuration changes is NFC_RegisterForConfigChange().
4. At any point of time, the default configuration can be restored by using NFC_SetDefaultConfig().

The configurable parameters are:

- Bluetooth passkey: No passkey, 1111, 5555 or any desired. (Default - No Passkey).
- Polling time out settings (in sec): Time the stack waits for a response from the controller before declaring Timeout - 30s, 60 or 90s (Default - 90).

✓ ***NOTE*** Before modifying the Configuration, first obtain the system configuration with properly populated fields using NFC_GetConfig() and then continue to modify the required fields. Subsequently set the configuration using NFC_SetConfig().

NFC Radio and Polling

The NFC radio will be turned on by the following APIs:

- **NFC_RegisterForTagDetection()**: When this function is called, the radio is turned on and polling is started for all the technologies registered.
- **NFC_RegisterForMessage()**: When this function is called, the radio is turned on and polling is started for all technologies.
- **NFC_EnableBTPairing()**: When this function is called, the radio is turned on and poling is started for all technologies.
- **NFC_EnableCE()**: When this function is called, the radio is turned on and Emulation Phase of Polling loop is started.

The polling and service layer activities:

1. The polling process starts and detects the tag in activity. The service layer receives a list of tags in proximity.
2. The service layer connects to the first tag in the list and checks if it is NDEF compliant and has an NDEF message.
3. If the tag has NDEF messages, it reads the NDEF messages (If multiple NDEF message support is available, all the messages would be read, check in Capabilities Multiple_NDEFMessages flag). For each NDEF message it reads in, invokes the callback registered for handling that message (TNF and type of the first record should have been registered). If multiple applications have registered for the same TNF and TYPE, all are invoked one after the other.
4. If there are no NDEF messages or if no callback has been registered for the NDEF message, the service layer checks if any application has registered for the technology. If yes, the application registered callbacks are invoked one after the other.
5. The above process is repeated for the next tag in the list. Prior to this it checks if the tag is still present in the RF field and continues with the process.
6. If there are no tags left in the list, the Polling/Discovery process is restarted.

The NFC radio will be turned off by the following APIs:

- **NFC_CloseHandle()**: When this function is called, with the handle obtained through NFC_RegisterForTagDetection() or NFC_RegisterForMessage(), the radio is turned off.
- **NFC_CloseAllHandles()**
- **NFC_DeInit()**
- **NFC_DisableBTPairing()**
- **NFC_DisableCE()**

✓ ***NOTE*** The radio is turned off if it were to be on.

The radio is turned off only after all the close requests are received corresponding to on requests. For example: if both NFC_RegisterForMessage() and NFC_EnableBTPairing() were called by the application, only after NFC_CloseHandle() and NFC_DisableBTPairing() are called, the radio is turned off.

Configuring Application Types and Modes

Application Type - Reading and writing NDEF Message(s)

NFC Data Exchange Format (NDEF) is a lightweight binary message format. It is designed to encapsulate one or more application payloads in to single message construct. The single message construct is called NDEF Message. Each NDEF message consists of one or more NDEF records. Each NDEF record can carry a payload of an arbitrary type and up to 2^31 - 1 octets in size. If the payload is larger, then the records can be chained to support bigger data.

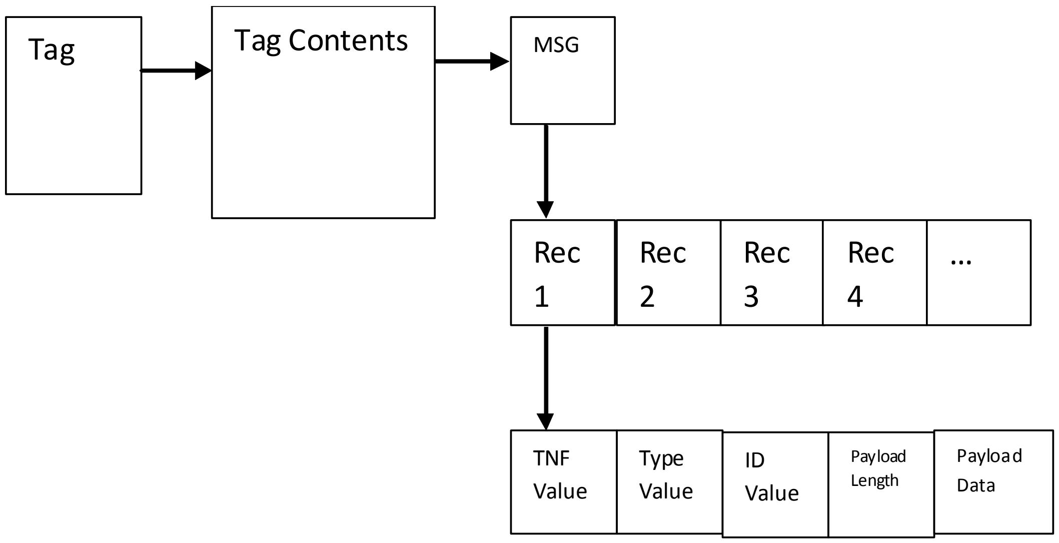

Figure 5-1 *NDEF Message Format*

The TNF values are:

1. TNF-WKT (Well-Known Type)
2. TNF-MIME
3. TNF-AbsURI
4. TNF-EXT (External Type)

5. TNF-UNK (Unknown Type)

The application will register the callback for handling the TNF value. The application can also filter for specific types within the TNF value, e.g. an application can register for TNF-EXT and type “xyz.com:propType” or an application can register for TNF-WKT and type “U”.

TYPE values for various TNF are described below:

TNF (0x01): NFC Forum Well known types(WKT)

An NDEF message contains the TYPE field. For a TNF type 0x01, NFC forum has defined the following Global types:

- Text Type: T
- URI Type: U
- Smart Poster Type: Sp
- Handover Request: Hr
- Handover Select: Hs
- Signature Type: Sig
- Generic Control: Gc

Local types can also be defined where the application can understand the context. E.g. a Smart Poster contains a record with WKT local type of “act”. Note that Global Types starts with an uppercase letter whereas a local type starts with a lower case letter.

TNF (0x02) MIME Type

Type example: image/jpeg

TNF (0x03) Absolute URI Type

Type: type field contains an absolute-URI BNF construct defined by RFC 3986

TNF (0x04) NFC Forum External Type (EXT)

External types should have the domain name of the organization defining the type, followed by the type name.

E.g. PropType defined by organization xyz has the following type name - xyz.com:PropType

Application Mode - Handling NFDEF Messages

The application needs to register for Handling NDEF messages by using NFC_RegisterForMessage(). Through the function NFC_RegisterForMessage(), the application can register for a specific TNF (Type Name Format)* value:

An application in one call to NFC_RegisterForMessage can register for only one TNF value, but can filter for multiple types. The types are provided to the function in an array and no. of types is also an input to the NFC_RegisterForMessage. An application can call NFC_RegisterForMessage several times to register multiple callbacks for different TNFs or same callback for different TNFs.

If the NFC radio is not on, a call to NFC_RegisterForMessage will turn NFC radio on and start the Polling/Discovery process.

If a tag is detected with first record of the NDEF message containing the TNF and TYPE value registered by the application, the callback is invoked.

The callback will provide the following parameters: Message handle, number of records in the message, TNF value, array of payload lengths, array of type lengths and array of ID lengths corresponding to each record. The application can now read the NDEF Records using NFC_GetRecord.

The application can call NFC_GetRecord and obtain the TNF, payload, TYPE and ID. If the payload in the record contains an embedded NDEF message, the same can be extracted by using NFC_GetEnclosedMessageInfo and subsequently NFC_GetRecord.

After handling the messages and records, the application calls NFC_CloseHandle to notify the service layer. The application can unregister by calling NFC_CloseHandle with the registry handle. If there are no other users of NFC, the NFC is turned off and the polling process is stopped.

Below is the sequence of steps (calls) the application needs to perform to retrieve and manage NDEF record(s):

1. NFC_Init()
2. NFC_Get Capabilities (); // API recommended to verify the stack capabilities.
3. NFC_RegisterForMessage()
   a. NFC radio will be automatically turned on and Polling/Discovery process starts.
4. As part of message callback the following information is received:
   a. Message Handle
   b. Number of Records in the Message
   c. TNF value
   d. Array of Length of Payloads
   e. Array of Type Lengths
   f. Array of ID Lengths
5. NFC_GetRecord() to obtain payload. This function returns:
   a. TNF value
   b. TYPE Value
   c. ID
   d. Payload.
6. Optionally, NFC_GetEnclosedMessageInfo() could be used to extract an embedded NDEF message and NFC_GetRecord()
7. NFC_CloseHandle()* to close Message Handles obtained.
8. NFC_CloseHandle()* to close handle received through callback - hOperation.
   a. If there are other Apps waiting for the same TNF and TYPE, those callbacks are invoked.
9. NFC_CloseHandle()* with the registry handle -
   a. If there is no other app waiting for NFC resources the NFC radio is turned off and the polling process is stopped.
10. NFC_DeInit() to de-initialize the stack.0

✓ ***NOTE*** * Calling NFC-DeInit() will close all open handles

Application Mode - Writing NDEF Messages

This application is concerned in writing NDEF message to tags of specific types. The application after calling NFC_Init() and optionally NFC_GetCapabilities() and configuration related APIs, registers for handling tags. The API used is NFC_RegisterForTagDetection.

Through the function NFC_RegisterForTagDetection, the application can register for specific tag types or all tag types.

If the NFC is not on, a call to NFC_RegisterForTagDetection will turn on NFC and also start the Polling/Discovery process.

The application preferably should have created NDEF message and be ready for writing before calling NFC_RegisterForTagDetection. The application can use NFC_CreateMessage to create a message, add records to it using NFC_AddRecord. Multiple records can be added by calling NFC_AddRecord using the same message handle. This message can be used in NFC_WriteMessage. NFC_WriteMessage takes inputs of messages and the no. of messages and writes into the connected tag. (currently only one message writing is supported).

When a tag type registered by the application is detected, its callback is invoked. The callback provides information on the tag in a structure. If bIsNDEF in the structure is TRUE, then the tag is NDEF formatted, if bIsLocked is FALSE, the tag is writable. If the tag is not NDEF formatted and formatting for that tag type is supported, the application can call NFC_NDEFFormat to format the tag. NFC_WriteMessage can be called to write in the NFC Message.

After handling the writing, the application calls NFC_CloseHandle to notify the service layer of the completion. The application can unregister by calling NFC_CloseHandle with the registry handle. If there are no other users of NFC, the NFC is turned off and the polling process is stopped.

Below is the sequence of steps (calls) the application needs to perform write NDEF message(s) to a tag:

1. NFC_Init()
2. NFC_GetCapabilities() // API recommended to verify the stack capabilities.
3. NFC_CreateMessage()
4. NFC_AddRecord() to add record into a message; this API can be called multiple times to add more records. (Record structure filled in with required record information and message handle are the inputs. See *NFC_Record on page 3-3*).
5. NFC_RegisterForTagDetection.
   a. Radio will be turned on.
6. Optionally: NFC_NDEFFormat()
7. NFC_WriteMessage() takes following inputs:
   a. Input of messages and the no. of messages.
   b. Callback will be invoked on completion with status.
8. NFC_CloseHandle()* with handles to the connection and message.
9. NFC_CloseHandle()* with the registry handle.
   a. If there is no other app waiting for NFC resources the NFC radio is turned off and the polling process is stopped.
10. NFC_DeInit() to de-initialize the stack.

✓ ***NOTE*** * Calling NFC-DeInit() will close all open handles

Application Mode - Enabling Read-Only

✓ ***NOTE*** This application type is a subset of Writing NDEF Message and is only enabling making Tag Read-Only.

Once a tag is made read-only, it cannot be modified/erased/written again.

If an application wants to add this functionality to the application's Writing NDEF Message, then the following additional steps are required.

Confirm the tag type in NFC_TagInfo structure obtained through callback is supporting 'read-only' capability. This information is available through NFC_GetCapabilities. After NFC_WriteMessage callback has confirmed the success of writing the message, Call NFC_MakeReadOnly.

Below is the sequence of steps (calls) the application needs to perform to make a tag read-only:

1. NFC_Init()
2. NFC_GetCapabilities() (it would provide confirmation if tag type supports Make read-only capability).
3. NFC_CreateMessage()
4. NFC_AddRecord() to add record into a message; this API can be called multiple times to add more records.

   (The inputs for Record Structure are: a record type information and the message handle. See *Chapter 3, Structures* for Record Structure description).
5. NFC_RegisterForTagDetection
   a. Radio will be turned on
   b. (Callback provides NFC_TagInfo structure data, which indicates Tag Type, the capability to make the Tag read only can be confirmed by checking with the structure NFC_CapStruct.TagsMakeReadOnlySupported obtained through NFC_GetCapabilities, similarly NDEF Formatting support is indicated through structure NFC_CapStruct.TagsFormattingSupported)
6. Optionally: NFC_NDEFFormat
7. NFC_WriteMessage() takes following inputs:
   a. Input of messages and the No. of messages.
8. NFC_MakeReadOnly() will enable tag read-only.
9. NFC_CloseHandle()* with handles to the connection and message.
10. NFC_CloseHandle()* with the registry handle.
    a. If there is no other app waiting for NFC resources the NFC radio is turned off and the polling process is stopped.
11. NFC_DeInit() - De-Initialize stack.

✓ ***NOTE*** * Calling NFC-DeInit() will close all open handles.

Application Mode - Enabling Card-Emulation

This application is concerned with card emulation. The application calls NFC_Init() and NFC_GetCapabilities(). NFC_GetCapabilities would provide information whether card emulation is supported and the type of card emulation supported. The application can call NFC_EnableCE(), with parameters of UICC/eSE/Host as the source for emulation, and a callback which will be invoked for any notifications. Callback can be NULL to disable notifications.

For Host based emulation, the application needs to call NFC_RegisterForHostCardEmulation() and NFC_AddRecord() before calling NFC_EnableCE().

If NFC is not on, it will be turned on and the Polling loop is set to Emulation phase only. If a callback was provided during registration, the callback will be invoked indicating the type of Transaction that has occurred.

The card emulation can be turned off by calling NFC_DisableCE(). If there are no other users of NFC, the NFC is turned off and the polling process is stopped.

Below is the sequence of steps (calls) the application needs to perform to enable Card Emulation:

1. NFC_Init()
2. NFC_GetCapabilities() (it would provide information if Card Emulation is supported).
3. Optionally for Host based Emulation: NFC_RegisterForHostCardEmulation() - parameters include:
   a. Size of Card
   b. Read Access
   c. Write Access
   d. Pointer to Message Handle.
4. Optionally for Host based Emulation: NFC_AddRecord(); zero or more records can be added to the message.
5. NFC_EnableCE() parameters includes:
   a. UICC/eSE
   b. Notification Callback Handler registration.

   NFC radio is turned on and emulation phase is included in the polling loop.
6. NFC_DisableCE() (It will disable Card Emulation - turned off).
   a. If there is no other app waiting for NFC resources the NFC radio is turned off and the polling process is stopped.
7. NFC_DeInit().

Application Mode - Enabling Bluetooth Pairing through NFC

This application is concerned with enabling Bluetooth pairing through NFC. Once the device is paired, it can use other Bluetooth APIs and continue. The application calls NFC_Init() and NFC_GetCapabilities(). NFC_GetCapabilities would provide information whether BT-Pairing is supported and the type of Bluetooth Pairing (STATIC/DYNAMIC). The pin key can be set through NFC_SetConfig(). The application can then call NFC_EnableBTPairing().

If NFC is not on, it will be turned on and the Polling loop is set to all tags/technologies. Even if NFC is on for only specific tags/technologies, this call enables all tags/technologies. If the notification is set, on any Bluetooth device paired, the callback will be invoked passing on a structure containing paired device details.

When NFC_DisableBTPairing() is called, and if there are no other applications/users, the NFC is turned off and the polling process is stopped.

Below is the sequence of steps (calls) the application needs to perform to enable Bluetooth Pairing:

1. NFC_Init()
2. NFC_GetCapabilities() (it would provide information if Bluetooth Pairing is supported).
3. NFC_GetConfig() //to get current configuration (pass key, polling timeout).
4. NFC_SetConfig() to set Pass Key.
5. NFC_RegisterForConfigChange() (Registers to get notified if any other App changes configuration).
6. NFC_EnableBTPairing() parameters included.
    a. Notification Callback Handler registration.

    (NFC Radio will be turned on, when a Static Handover Message is received, Bluetooth radio is turned on, if it is off and Pairing/Bonding is initiated. On completion callback is invoked).
7. NFC_DisableBTPairing() (Disables Bluetooth Pairing through NFC).
    a. If there is no other app waiting for NFC resources the NFC radio is turned off and the polling process is stopped.
8. NFC_DeInit().

Application Mode - Direct Reading/Writing over NFC

This application is concerned with transacting a proprietary protocol over layer like ISO14443A-3 for reading/writing raw data. The application registers for tags with the protocol supported with NFC_RegisterForTagDetection(). On callback, it can check with tag infostructure, the protocols supported by the tag. The application can then use NFC_ExchangeData(), to send its command and data, and receive the response and data from the tag.

If NFC is not on, it will be turned on and the polling loop is set to the required tags/technologies.

Below is the sequence of steps (calls) the application needs to perform transactions over lower Protocol:

1. NFC_Init().
2. NFC_GetCapabilities() // Verify the stack capabilities using this API.
3. NFC_RegisterForTagDetection() (TagInfo is received in Callback, when a tag is detected).
4. NFC_ExchangeData (can be repeated many times).
5. NFC_CloseHandle().
6. NFC_DeInit().

Example 1:

This example is providing steps for reading/writing to MiFare Ultralight tag directly using commands and data. The parameters used in NFC_ExchangeData for Reading/Writing raw data to MiFare Ultralight are given below:

For writing 4 bytes to Mifare ultralight:

> Char pReaderToTagBuffer[] = {0xA2, 0x04, 0x2A, 0x2B, 0x2C, 0x2D, 0xc0, 0xc9};
> The first 6 elements of the buffer:
> 0xA2: “Write” command identifier for Mifare Ultralight.
> 0x04: Block No. to be written.
> 0x2A, 0x2B, 0x2C, 0x2D: data Bytes
> Last two bytes are CRC bytes.

For writing total 48 bytes:

Since MiFare Ultralight command set supports writing only 4 bytes at a time, NFC_ExchangeData() has to be called 12 times, each time with Block No., the corresponding data for the block and CRC bytes in pReaderToTagBuffer[] for 48 bytes to be written.

For reading:

char pReaderToTagBuffer[] = {0x30, 0x04, 0x26, 0xee};

0x30: “Read” command identifier for Mifare Ultralight.

0x04: Block No. from where data to be read.

Last two bytes are CRC bytes.

Read data will be available in “pTagToReaderBuffer”.

NDEF Message

Smart Poster NDEF Message Creation Example

A smart poster NDEF message is created with the following data:

1. URL: http://www.msi.com/mc45/.
2. Text: “A Tour of MC45.”
3. Action: Launch browser.

A smart poster message is structured as follows:

1. Sp NDEF message contains only one record- of TNF value - TNF-WKT and Type: Sp.
2. The payload of the Sp record contains in this example three records: URI, text and action records.

The creation of the Smart poster message is as follows.

1. First create a message (This message goes as a payload into Smart Poster Record)
2. Add the three records (URI, text and action) into this message.
3. Get a copy of the NDEF message created.
4. Create a message (This will be the Message written into tag).
5. Add an Sp record with the payload being the NDEF message obtained in (3).

Below are the details:

1. Enclosed Message Creation

   NFC_Handle enclMessageHandle;// Handle to the enclosed Message

   NFC_CreateMessage(&enclMessageHandle); // Create enclosed Message

2. Records Creation

   URI Record Creation:

```
UNS8 Buffer[] = “msi.com/mc45”;

NFC_Record NDEFRecord;

NDEFRecord.Type = TYPE_URI;

NDEFRecord.URI.URIType = HTTP_WWW;

NDEFRecord.URI.Payload = Buffer;

NFC_AddRecord(NDEFRecord, &enclMessageHandle); // Use this API to add the record to the
message.
```

   Text Record Creation:

```
UNS8 Buffer[] = “A Tour of MC45.”

UNS8 lang[] = “en”

NFC_Record NDEFRecord;

NDEFRecord.Type = TYPE_TEXT;

NDEFRecord.Text.CharEncoding = CHAR_UNS8;

NDEFRecord.Text.Lang = lang;

NDEFRecord.Text.Payload = Buffer;

NDEFRecord.Text.PayloadLength = strlen(Buffer);

NFC_AddRecord(NDEFRecord, &enclMessageHandle);
```

   Action Record Creation:

```
UNS8 Buffer = “0”; // 0 indicating “launch the URI”

UNS8 type[] = “act”;

NFC_Record NDEFRecord;

NDEFRecord.Type = TYPE_GENERAL;

NDEFRecord.General.TNF = TNF_WKT;

NDEFRecord.General.Type = type;

NDEFRecord.General.IDLength = 0;

NDEFRecord.General.Payload = Buffer;

NDEFRecord.General.PayloadLength = strlen(Buffer);

NFC_AddRecord(NDEFRecord, &enclMessageHandle);
```

3. Get Copy of enclosed Message

```
UNS8 Buffer[100];

DWORD BufLength;

NFC_GetMessageContent(enclMessageHandle, Buffer, &BufLength);
```

4. Smart Poster Message Creation

   NFC_Handle spMessageHandle;

```
NFC_CreateMessage(&spMessageHandle);
```

5. Smart Poster Record Addition

```
NFC_Record NDEFRecord;

NDEFRecord.Type = TYPE_GENERAL;
NDEFRecord.General.TNF = TNF_WKT;
NDEFRecord.General.Type = type;
NDEFRecord.General.IDLength = 0;
NDEFRecord.General.Payload = Buffer;
NDEFRecord.General.PayloadLength = BufLength;
NFC_AddRecord(NDEFRecord, &enclMessageHandle);
```

Now, message can be written as follows:

```
Handles[0]=spMessageHandle;
NFC_WriteMessage(hConnection, Handles, 1);
```

Below is the sequence of steps (calls) the application needs to perform to write Smart Poster NDEF Message:

1. NFC_Init()
2. NFC_GetCapabilities() (it would provide confirmation if Tag type supports Make read-only capability)
3. NFC_CreateMessage() for Enclosed message creation.
4. NFC_AddRecord() to add URI record.
5. NFC_AddRecord() to add text record.
6. NFC_AddRecord() to add action record.
7. NFC_GetMessageContent(enclMessageHandle1, Buffer, &BufLength);
8. NFC_CreateMessage() for Smart Poster Message creation.
9. NFC_AddRecord() for Smart Poster record addition.
10. NFC_RegisterForTagDetection.
    a. Radio will be turned on,
11. (Callback - receive handle to the tag).
12. Optionally: NFC_NDEFFormat.
13. NFC_WriteMessage(hConnection, Handles, 1); //Handles[0]=spMessageHandle
14. Optionally: NFC_MakeReadOnly() to make tag read-only.
15. NFC_CloseHandle(); // Issue close handle for the connection handle received through callback. The application can wait for further callbacks and continue to write messages on other Tags by repeating steps 11 to 15 OR If the application is done it can go through further steps 16 & 17 and exit.
16. NFC_CloseHandle(); // Optionally, Issue close Handles for messages and registry.

   a. If there is no other app waiting for NFC resources the NFC radio is turned off and the polling process is stopped.

17. NFC_DeInit(); // will close all open handles.

Smart Poster NDEF Message Reading Example

This example shows the reading of the above Smart Poster NDEF message.

The reading of the Smart poster message is as follows:

1. As part of the callback API, receive the number of records in the message and the TNF type. This would show 1 and TNF-WKT respectively.
2. Read the first record and Get the Type, It shows “Sp” indicating the payload has an NDEF Message.
3. Using NFC_GetEnclosedMessageInfo(), read the Message handle of the enclosed message, number of records, TNF, payload lengths, type lengths and ID lengths.
4. Using the NFC_GetRecord, read the individual records.

First Message Handling

```
// Callback API indicates noOfRecords is one and tnf is TNF-WKT and provides messageHandle;
```

Reading Record of the main message

```
    UNS8 Buffer[100];
    WORD tnf;
    UNS8 type[10];
NFC_GetRecord(messageHandle, 1, &tnf, Buffer, type, NULL);
//type will indicate that this is Sp record
```

Reading enclosed message info in the SP Record

```
    DWORD PayloadLengths[10];
    WORD TypeLengths[10], IDLengths[10], noOfRecords;
    NFC_HANDLE enclMessageHandle;
NFC_GetEnclosedMessageInfo(MessageHandle, 1, &enclMessageHandle, &noOfRecords, PayloadLengths,
TypeLengths, IDLengths)
// the enclosed message has 3 records
```

Reading individual records in the enclosed message

```
    UNS8 Buffer1[100], Buffer2[100],Buffer3[100];
    WORD tnf1, tnf2, tnf3;
    UNS8 type1[10], type2[10], type3[10];

    NFC_GetRecord(enclMessageHandle, 1, &tnf1, Buffer1, type1, NULL);
    NFC_GetRecord(enclMessageHandle, 2, &tnf2, Buffer2, type2, NULL);
    NFC_GetRecord(enclMessageHandle, 3, &tnf3, Buffer3, type3, NULL);
```

Below is the sequence of steps (calls) the application needs to perform to read Smart Poster NDEF Message:

1. NFC_Init()
2. NFC_RegisterForMessage(appCallback, NULL, TNF-WKT, Typesarray, 1, &regMsgHandler).
   a. NFC radio will be automatically turned on - Polling/Discovery process starts
3. Need to wait for the callback MessageHandle, No. of Records, tnf, PayloadLengths[], TypeLengths[], IDLengths[] will be available through callback
4. NFC_GetRecord(MessageHandle, 1, &tnf, Buffer, type, NULL); // since we have registered for only "Sp" message above, this step is not necessary. (With this step we are seeing the record type is "Sp" and continuing to GetEnclosedMessageInfo)
5. NFC_GetEnclosedMessageInfo(MessageHandle, 1, enclMessageHandle &noOfRecords, &tnf, PayloadLengths, TypeLengths, IDLengths)
6. NFC_GetRecord(enclMessageHandle, 1, &tnf1, Buffer1, type1, NULL);
7. NFC_GetRecord(enclMessageHandle, 2, &tnf2, Buffer2, type2, NULL);
8. NFC_GetRecord(enclMessageHandle, 3, &tnf3, Buffer3, type3, NULL);
9. NFC_CloseHandle(); // Issue close handles for the message handles and connection handle received through callback. The application can wait for further callbacks and continue to receive and handle further messages by repeating steps 4 to 9 or If the application is done it can go through further steps 10 and 11 and exit.
10. NFC_CloseHandle(); //optionally, with the registry handle.
    a. If there is no other app waiting for NFC resources the NFC radio is turned off and the polling process is stopped.
11. NFC_DeInit(); // will close all open handles.

Zebra Technologies Corporation
Lincolnshire, IL U.S.A
http://www.zebra.com

MN000145A02 Revision A - February 2015