MagTek Universal SDK

Programmer's Manual (Linux)

February 2024

Manual Part Number:
D998200570-100

REGISTERED TO ISO 9001:2015

Table 0.1 – Revisions

Rev Number	Date	Notes
100	February 1, 2024	Initial Release.

SOFTWARE LICENSE AGREEMENT

IMPORTANT: YOU SHOULD CAREFULLY READ ALL THE TERMS, CONDITIONS AND RESTRICTIONS OF THIS LICENSE AGREEMENT BEFORE INSTALLING THE SOFTWARE PACKAGE. YOUR INSTALLATION OF THE SOFTWARE PACKAGE PRESUMES YOUR ACCEPTANCE OF THE TERMS, CONDITIONS, AND RESTRICTIONS CONTAINED IN THIS AGREEMENT. IF YOU DO NOT AGREE WITH THESE TERMS, CONDITIONS, AND RESTRICTIONS, PROMPTLY RETURN THE SOFTWARE PACKAGE AND ASSOCIATED DOCUMENTATION TO THE ADDRESS ON THE FRONT PAGE OF THIS DOCUMENT, ATTENTION: CUSTOMER SUPPORT.

TERMS, CONDITIONS, AND RESTRICTIONS

MagTek, Incorporated (the "Licensor") owns and has the right to distribute the described software and documentation, collectively referred to as the "Software."

LICENSE: Licensor grants you (the "Licensee") the right to use the Software in conjunction with MagTek products. LICENSEE MAY NOT COPY, MODIFY, OR TRANSFER THE SOFTWARE IN WHOLE OR IN PART EXCEPT AS EXPRESSLY PROVIDED IN THIS AGREEMENT. Licensee may not decompile, disassemble, or in any other manner attempt to reverse engineer the Software. Licensee shall not tamper with, bypass, or alter any security features of the software or attempt to do so.

TRANSFER: Licensee may not transfer the Software or license to the Software to another party without the prior written authorization of the Licensor. If Licensee transfers the Software without authorization, all rights granted under this Agreement are automatically terminated.

COPYRIGHT: The Software is copyrighted. Licensee may not copy the Software except for archival purposes or to load for execution purposes. All other copies of the Software are in violation of this Agreement.

TERM: This Agreement is in effect as long as Licensee continues the use of the Software. The Licensor also reserves the right to terminate this Agreement if Licensee fails to comply with any of the terms, conditions, or restrictions contained herein. Should Licensor terminate this Agreement due to Licensee's failure to comply, Licensee agrees to return the Software to Licensor. Receipt of returned Software by the Licensor shall mark the termination.

LIMITED WARRANTY: Licensor warrants to the Licensee that the disk(s) or other media on which the Software is recorded are free from defects in material or workmanship under normal use.

THE SOFTWARE IS PROVIDED AS IS. LICENSOR MAKES NO OTHER WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Because of the diversity of conditions and PC hardware under which the Software may be used, Licensor does not warrant that the Software will meet Licensee specifications or that the operation of the Software will be uninterrupted or free of errors.

IN NO EVENT WILL LICENSOR BE LIABLE FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE, OR INABILITY TO USE, THE SOFTWARE. Licensee's sole remedy in the event of a defect in material or workmanship is expressly limited to replacement of the Software disk(s) if applicable.

GOVERNING LAW: If any provision of this Agreement is found to be unlawful, void, or unenforceable, that provision shall be removed from consideration under this Agreement and will not affect the enforceability of any of the remaining provisions. This Agreement shall be governed by the laws of the State of California and shall inure to the benefit of MagTek, Incorporated, its successors or assigns.

ACKNOWLEDGMENT: LICENSEE ACKNOWLEDGES THAT HE HAS READ THIS AGREEMENT, UNDERSTANDS ALL OF ITS TERMS, CONDITIONS, AND RESTRICTIONS, AND AGREES TO BE BOUND BY THEM. LICENSEE ALSO AGREES THAT THIS AGREEMENT SUPERSEDES ANY AND ALL VERBAL AND WRITTEN COMMUNICATIONS BETWEEN LICENSOR AND LICENSEE OR THEIR ASSIGNS RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

QUESTIONS REGARDING THIS AGREEMENT SHOULD BE ADDRESSED IN WRITING TO MAGTEK, INCORPORATED, ATTENTION: CUSTOMER SUPPORT, AT THE ADDRESS LISTED IN THIS DOCUMENT, OR E-MAILED TO SUPPORT@MAGTEK.COM.

DEMO SOFTWARE / SAMPLE CODE: Unless otherwise stated, all demo software and sample code are to be used by Licensee for demonstration purposes only and MAY NOT BE incorporated into any production or live environment. The PIN Pad sample implementation is for software PIN Pad test purposes only and is not PCI compliant. To meet PCI compliance in production or live environments, a third-party PCI compliant component (hardware or software-based) must be used.

Table of Contents

1 Introduction

This document provides instructions for software developers who want to create software solutions that include a DynaWave, mDynamo, DynaProx, DynaFlex Pro, and DynaFlex II PED connected to a Linux-based host via USB. It is part of a larger library of documents designed to assist MagTek devices implementers, which includes the following documents available from MagTek:

The following documents are essential:

- ***D99875475 MagneSafe V5 Communication Reference Manual***
- ***D998200383 DynaFlex Products Programmer's Manual (COMMANDS)***

1.1 About the MagTek Universal SDK C++ (Linux)

The MTUSDK C++ (Linux), available from MagTek, provides demonstration source code and a reusable MTUSDK library that provide developers of custom software solutions with an easy-to-use interface for DynaWave, mDynamo, DynaProx, DynaFlex Pro, and DynaFlex II PED. Developers can include the MTUSDK library in custom branded software which can be distributed to customers or distributed internally as part of an enterprise solution.

1.2 Nomenclature

The general terms "device" and "host" are used in different, often incompatible ways in a multitude of specifications and contexts. For example "host" may have different meanings in the context of USB communication than it does in the context of networked financial transaction processing. In this document, "device" and "host" are used strictly as follows:

- **Device** refers to the device (eg. DynaProx) that receives and responds to the command set specified in this document.
- **Host** refers to the piece of general-purpose electronic equipment the device is connected or paired to, which can send data to and receive data from the device. Host types include PC and Mac computers/laptops, tablets, smartphones, teletype terminals, and even test harnesses. In many cases the host may have custom software installed on it that communicates with the device. When "host" must be used differently, it is qualified as something specific, such as "USB host."

The word "user" is also often used in different ways in different contexts. In this document, user generally refers to the cardholder.

1.3 SDK Contents

File	Description
mtscra	Folder containing the libmtscra.a library source code.
mtmms	Folder containing the libmtmms.a library source code.
mtscratest	Folder containing the MTSCRA Test console app.
mtusdk	Folder containing the MTUSDK library source code..
mtusdk_test	Folder containing the MTUSDK Test app for EMV transaction.

1.4 System Requirements

Tested operating systems:
Ubuntu 20.04.5

Tested development environments:
Windows 10 with Microsoft Visual Studio 2017

1.5 Interfaces for Operating Systems

The following table matches the device interface to operating system.

Device	Interface	Operating System
DynaFlex II PED	USB	Ubuntu
DynaFlex Pro	USB	Ubuntu
DynaProx	USB Serial	Ubuntu
mDynamo	USB Serial	Ubuntu
DynaWave	USB Serial	Ubuntu

2 How to Set Up the MTUSDK Libraries

2.1 How to Install Dependencies

1) Install libjsoncpp-dev for json api

```
sudo apt install libjsoncpp-dev
```

2) Install libcurl for http/https access.

```
sudo apt install libcurl4-openssl-dev
```

3) Install restclient-cpp.

```
git clone https://github.com/mrtazz/restclient-cpp
```

```
sudo apt install libtool
```

```
cd restclient-cpp
```

```
./autogen.sh
```

(installs to usr/local/lib)

```
sudo make install
```

4) Install libusb.

```
sudo apt-get install libusb-1.0-0-dev
```

5) Install lissl-dev for base64 encoding/decoding.

```
sudo apt install libssl-dev
```

6) Cryptopp from https://github.com/weidai11/cryptopp. Cryptopp is automatically downloaded during the make build of mtusdk static library. Cryptopp is a static library to implement hash sha256.

2.2 How to Build

2.2.1 Build libraries

1) To build libraries:
   a. Extract mtusdk_linux.zip.
   b. Run the make command.

```
make
```

2) The libraries are built and sent to the mtusdk_test folder.
   - libmtmms.so (shared object)
   - libmtscra.so (shared object)
   - libmtusdk.a (static library)

2.2.2 Rebuild MTUSDK Test app

1) To rebuild the MTUSDK Test app after any modifications.
   a. Navigate to mtusdk_test folder.
   b. Run the make command.

```
make
```

2.2.3 Run the MTUSDK Test app

1) To run the MTUSDK Test app:

```
sudo ./mtusdk_test.mms
```

2) The response is similar as below:

```
SDK version 118
   Model:
   Name: B62CA5F0
 <- (Connection State) connected
 <- (Operation Status) operation_done,0000D101,Operation
Done,00000000
 <- (Device Response) AA...
   Serial Number: B62CA5F0
Select an operation :
 1. Transaction
 2. Send configuration file
 3. Update firmware
 4. Display a message
 5. Send an image and display
 6. Setup device driven fallback
 7. Setup host driven fallback
 8. Enable event driven transaction
 9. Disable event driven transaction
 a. Update key to Magensa key
 0. Quit
```

3) If the command is not successful but is similar to:

```
librestclient-cpp.so cannot open shared object file no such file or
directory.
```

then send the following command to link the shared object file. Retry after this.

```
sudo /sbin/ldconfig
```

3 CoreAPI

Use the CoreAPI to create an IDevice. IDevice is the bases for the MagTek Universal SDK.

3.1 getDevice

This function returns an instance of a device.

```
IDevice* getDevice(
      MTU_DEVICE_TYPE DeviceType,
      MTU_DEVICE_CONNECTION_TYPE ConnectionType,
      string DeviceAddress);

IDevice* getDevice(
      MTU_DEVICE_TYPE DeviceType,
      MTU_DEVICE_CONNECTION_TYPE ConnectionType,
      string DeviceAddress,
      CertificateInfo cert);
```

Parameter	Description
DeviceType	Enumerated device type.
ConnectionType	Enumerated connection type.

Parameter	Description

DeviceAddress	Address for the device.
	
	For USB devices, address may be an empty string when only one device is attached. Otherwise address should be in the form:
	`USB://DEVICESERIALNUMBER`
	for example, `USB://99261829170E0810`
	
	For Ethernet devices, address should be in the form:
	`IP://IP-Address:PORT`
	for example, `IP://10.57.10.180:26`
	
	For Wireless DynaPro Go devices, address should be in the form:
	`TLS12://TLSDEVICESERIALNUMBER`
	`TLS12TRUST://TLSDEVICESERIALNUMBER`
	for example, `TLS12://TLS99261829170E0810`
	`TLS12TRUST://TLS99261829170E0810`
	
	For Bluetooth LE devices, address should be in the form:
	`BLEEMV://DEVICENAME`
	for example, `BLEEMV://DynaPro Go-EB66`
	
	For WebSocket or Secure WebSocket devices, address should be in the form:
	`ws://IP-Address or DEVICEADDRESS`
	`wss://IP-Address or DEVICEADDRESS`
	for example, `ws://10.57.10.1 or ws://b512345.magtek.com`
	`wss://192.168.1.150 or wss://b512345.magtek.com`
	
	For Serial devices, address should be in the form:
	PORT=**[PORT]**,
	BAUDRATE=**[BAUDRATE]**,
	DATABITS=**[DATABITS]**,
	PARITY=**[PARITY]**,
	STOPBITS=**[STOPBITS]**,
	HANDSHAKE=**[HANDSHAKE]**,
	STARTINGBYTE=**[STARTINGBYTE]**,
	ENDINGBYTE=**[ENDINGBYTE]**,
	CRCMODE=**[CRCMODE]**

cert	Certificate information for TLS Trust and WebSocket WSS connection. Optional. See the following for details on installing a certificate chain: ***D998200550 DYNAFLEX TLS CERTIFICATE INSTALLATION MANUAL***

Return Value: `IDevice*`.

3.2 getDeviceList

This function retrieves a list of connected devices.

```
vector<IDevice*> CoreAPI::getDeviceList();

vector<IDevice*> CoreAPI::getDeviceList(
      vector<MTU_DEVICE_TYPE> deviceTypes);

vector<IDevice*> CoreAPI::getDeviceList(
      MTU_DEVICE_TYPE deviceType);

vector<IDevice*> CoreAPI::getDeviceList(
      MTU_DEVICE_TYPE deviceType,
      MTU_DEVICE_CONNECTION_TYPE connectionType);
```

Parameter	Description
deviceType	Enumeration of MagTek readers devie types.
deviceTypes	Enumerated list of MagTek readers devie types.
connectionType	Enumerated connection type.

Return Value: `vector<IDevice*>`

3.3 getConnectionTypes

This function returns a list of conection types supported from a device type.

```
vector<MTU_DEVICE_CONNECTION_TYPE> CoreAPI::getConnectionTypes(
      MTU_DEVICE_TYPE deviceType);
```

Parameter	Description
deviceType	Enumerated device type.

Return Value: `vector<MTU_DEVICE_CONNECTION_TYPE>.`

3.4 getConnectionTypeString

This function returns a string for the connection type.

```
string CoreAPI::getConnectionTypeString(
      MTU_DEVICE_CONNECTION_TYPE connectionType);
```

Parameter	Description
connectionType	Enumerated connection type.

Return Value: A string for the connection type.

3.5 getConnectionTypeFromString

This function retrieves the connection type from a string.

```
MTU_DEVICE_CONNECTION_TYPE CoreAPI::getConnectionTypeFromString(
      string connectionTypeString);
```

Parameter	Description
connectionTypeString	String value for a connection type.

Return Value: `MTU_DEVICE_CONNECTION_TYPE`.

3.6 getAPIVersion

This function returns the API version.

```
int CoreAPI::getAPIVersion();
```

Return Value: An integer representing the API version.

3.7 isEnumerable

This function checks whether the device and connection type is enumerable.

```
bool CoreAPI::isEnumable(
      MTU_DEVICE_TYPE deviceType,
      MTU_DEVICE_CONNECTION_TYPE connectionType);
```

Parameter	Description
deviceType	Enumerated device type.
connectionType	Enumerated connection type.

Return Value: True if enumerable. Otherwise is false.

3.8 releaseDevice

This function releases a device.

```
void CoreAPI::releaseDevice(IDevice* Device);
```

Parameter	Description
Device	An IDevice.

Return Value: None

3.9 releaseDeviceList

This function releases a list of devices.

```
void CoreAPI::releaseDeviceList(vector<IDevice*>& devices);
```

Parameter	Description
devices	List of IDevice.

Return Value: None

4 IDevice

Create an instance of the **IDevice** from CoreAPI::getDeviceList(). Then use the functions described in this chapter.

4.1 getName

This function retrieves the name of the device.

```
IDeviceControl* IDevice->getName()
```

Return Value: `IDeviceControl*`

4.2 getConnectionInfo

This function retrieves the connection info of the device.

```
ConnectionInfo IDevice->getConnectionInfo()
```

Return Value: `ConnectionInfo`

4.3 getConnectionState

This function retrieves the connection state of the device.

```
ConnectionState IDevice->getConnectionState()
```

Return Value: `ConnectionState`

4.4 getDeviceInfo

This function retrieves the information class of the device.

```
DeviceInfo IDevice->getDeviceInfo()
```

Return Value: `DeviceInfo`

4.5 getDeviceCapabilities

This function retrieves the device capabilities class of the device.

```
DeviceCapability IDevice->getDeviceCapabilities()
```

Return Value: `DeviceCapability`

4.6 getDeviceControl

This function retrieves the device control class of the device.

```
IDeviceControl* IDevice->getDeviceControl()
```

Return Value: `DeviceCapability`

4.7 getConfiguration

This function retrieves the device configuration class of the device.

```
IDeviceConfiguration* IDevice->getConfiguration()
```

Return Value: `IDeviceConfiguration*`

4.8 subscribeAll

This function allows the host to be notified of all events sent by the device.

```
bool IDevice->subscribeAll(IEventSubscriber* eventCallback)
```

Parameter	Description
eventCallback	Name of a class that implements the IEventSubscriber interface event.

Return Value: Truth of successful. Otherwise, returns false.

4.9 unsubscribeAll

This function allows the host to no longer receive any events sent by the device.

```
bool IDevice->unsubscribeAll(IEventSubscriber* eventCallback)
```

Parameter	Description
eventCallback	Name of a class that implements the IEventSubscriber interface event.

Return Value: Truth of successful. Otherwise, returns false.

4.10 startTransaction

This function starts a transaction. This function will automatically handle the opening and closing of a device. The transaction will be processed through multiple calls to the event Error! Reference source not found.**()**. See **API Walk Through** for how to process an EMV transaction.

```
bool IDevice->startTransaction(MTUSDK_ITransaction& transaction)
```

Parameter	Description
eventCallback	Name of a class that implements the IEventSubscriber interface event.

Return Value: Truth of successful. Otherwise, returns false.

4.11 cancelTransaction

This function cancels a transaction. A transaction can only be canceled before a card is presented.

```
bool IDevice->cancelTransaction();
```

Return Value: True if canceled. Otherwise, returns false.

4.12 sendSelection

This function sends a user selection to the device. The response data will be returned in the event Error! Reference source not found..

```
bool IDevice->sendSelection(IData data);
```

Parameter for data	Description
Byte 0	Status of User Selection: 0x00 = User Selection Request completed, see Selection Result 0x01 = User Selection Request aborted, cancelled by user 0x02 = User Selection Request aborted, timeout
Byte 1	The menu item selected by the user. This is a single byte zero based binary value.

Return Value: True if successful. Otherwise, returns false.

4.13 sendAuthorization

This function sends the Authorization Response Code (ARPC) blob to the device. The response data will be returned in the event OnEvent(). See **API Walk Through** for how to process an EMV transaction.

```
bool IDevice->sendAuthorization(IData data);
```

Parameter	Description
data	Contains ARPC blob.

Return Value: True if successful. Otherwise, returns false.

4.14 requestPAN

This function prompts the user to present their card and enter a PIN. A card is presented so that the device can retrieve the PAN, which is used for Format blocks requiring a PAN. The encrypted PIN block (EPB) will be returned in the event OnEvent().

For DynaFlex devices, this function starts a PIN session on the first call and shall be called again to send the PIN status to the device for completing the PIN session.

```
bool IDevice->requestPAN(
      MTUSDK_PANRequest Request,
      MTUSDK_PINRequest* pinRequest = NULL);
```

Request:

Parameter	Type	Description
Timeout	unsigned char	Wait time in seconds.

Parameter	Type	Description
PaymentMethods	enum	PaymentMethod enumeration. Usage: `MSR = 1` - For magnetic stripe cards. `Contact = 2` - For EMV chip cards. `Contactless = 4` - For NFC contactless cards. `ManualEntry = 8` - For user to manually enter transaction data without any card access. When set to ManualEntry, the other payment methods do not apply. `BARCODE = 16` - For barcode transaction. `APPLE_VAS = 32` - For Apple VAS support.

`PINRequest:`

Parameter	Type	Description
Timeout	unsigned char	Wait time in seconds.
PINMode	unsigned char	PIN mode. Usage: `0x00` - Enter PIN `0x01` - Enter PIN Amount `0x02` - Reenter PIN Amount `0x03` - Reenter PIN `0x04` - Verify PIN For DynaFlex devices this is the User Interface Sequence: `0x01` - Present Card / Enter PIN (start session) `0x04` - Present Card / Enter PIN / Enter PIN Again (start session) On the second call to requestPAN(), send the PIN status: `0xFD` - Cancel PIN Session (end session) `0xFE` - PIN Entry Failed (end session) `0xFF` - PIN Entry Successful (end session)
MinLength	unsigned char	Minimum length of accepted PIN (>= 4).
MaxLength	unsigned char	Maximum length of accepted PIN (=< 12).
Tone	unsigned char	Tone to play when prompting for the PIN. Usage: `0x00` - No sound `0x01` - One beep `0x02` - Two beeps

Parameter	Type	Description
Format	unsigned char	ISO format for the PIN block. `0x00` - ISO Format 0 `0x01` – Reserved / Invalid `0x03` - ISO Format 3 `0x04` - ISO Format 4
PAN	string	First 12 digits of the Primary Account Number. Leave blank if not required by the ISO format for the PIN block.

Return Value: True if successful. Otherwise, returns false.

4.15 requestPIN

This function prompts the user to enter a PIN. The host must supply the PAN if the Format block selected requires a PAN. The encrypted PIN block (EPB) will be returned in the event OnEvent().

For DynaFlex devices, this function starts a PIN session on the first call and shall be called again to send the PIN status to the device for completing the PIN session.

```
bool IDevice->requestPIN(MTUSDK_PINRequest Request);
```

`Request:`

Parameter	Type	Description
Timeout	unsigned char	Wait time in seconds.
PINMode	unsigned char	PIN mode. Usage: `0x00` - Enter PIN `0x01` - Enter PIN Amount `0x02` - Reenter PIN Amount `0x03` - Reenter PIN `0x04` - Verify PIN For DynaFlex devices this is the User Interface Sequence: `0x00` - Enter PIN (start session) `0x02` - PIN Incorrect, Try Again (continue session) `0x03` - Enter PIN / Enter PIN Again (start session) `0x05` - Enter PIN Again (continue session) On the second call to requestPIN(), send the PIN status: `0xFD` - Cancel PIN Session (end session) `0xFE` - PIN Entry Failed (end session) `0xFF` - PIN Entry Successful (end session)
MinLength	unsigned char	Minimum length of accepted PIN (>= 4).

Parameter	Type	Description
MaxLength	unsigned char	Maximum length of accepted PIN (=< 12).
Tone	unsigned char	Tone to play when prompting for the PIN. Usage: `0x00` - No sound `0x01` - One beep `0x02` - Two beeps
Format	unsigned char	ISO format for the PIN block. `0x00` - ISO Format 0 `0x01` - ISO Format 1 `0x03` - ISO Format 3 `0x04` - ISO Format 4
PAN	string	First 12 digits of the Primary Account Number. Leave blank.

Return Value: True if successful. Otherwise, returns false.

4.16 requestSignature

This function prompts the user to enter a signature. The response data will be returned in the event OnEvent().

```
bool IDevice->requestSignature(unsigned char timeout);
```

Parameter	Description
timeout	Time in seconds for the operation to complete. 0x01 – 0xFF

Return Value: True of successful. Otherwise, returns false.

4.17 sendNFCCommand

This function sends a command to an NFC tag type 2. The NFC tag must first be activated by calling startTransaction() with NFC enabled.

```
bool IDevice-> sendNFCCommand(
      IData data,
      bool lastCommand,
      bool encrypt);
```

Parameter	Description
data	Command to send to the NFC tag. • `Get Version` • `Read` • `Fast Read` • `Write` • `Compatibility Write` • `Read_Cnt` • `PWD_Auth` • `Read_Sig` For details of the command see the NFC commands table below or ***D998200383 DynaFlex Family Programmer's Manual (COMMANDS) section NFC/Mifare Pass Through Commands***
lastCommand	Determines if this is the last NFC command to complete the operation. `true` = This is the last command. Device will provide a single beep after receiving a successful response from the NFC tag. To send subsequent commands, the NFC tag must be activated by calling startTransaction() with NFC enabled. `false` = Expect more commands (Default). Either set to true or false, if the NFC tag command fails, device will provide a double beep.
encrypt	Determines if data returned is to be encrypted. `true` = Encrypt data `false` = Do not encrypt data (Default)

Return Value: True of successful. Otherwise, returns false.

NFC Commands

Command	Length	Field Value
Get Version	1	The GET_VERSION command is used to retrieve information on the NTAG family, the product version, storage size and other product data required to identify the specific NTAG21x. Byte 0 = 0x60

Command	Length	Field Value
Read	2	The READ command requires a start page address, and returns the 16 bytes of four NTAG21x pages. For example, if address is 03h then pages 03h, 04h, 05h, 06h are returned. Special conditions apply if the READ command address is near the end of the accessible memory area. The special conditions also apply if at least part of the addressed pages is within a password protected area. Byte 0 = 0x30 Byte 1 = Start Page Address
Fast Read	3	The FAST_READ command requires a start page address and an end page address and returns the all n*4 bytes of the addressed pages. For example, if the start address is 03h and the end address is 07h then pages 03h, 04h, 05h, 06h and 07h are returned. Byte 0 = 0x3A Byte 2 = Start Page Address Byte 3 = End Page Address
Write	6	The WRITE command requires a block address, and writes 4 bytes of data into the addressed NTAG21x page. Byte 0 = 0xA2 Byte 1 = Address to Write Byte 2 to 5 = 4 Bytes of Data to Write
Compatibility Write	18	The COMPATIBILITY_WRITE command is implemented to guarantee interoperability with the established MIFARE Classic PCD infrastructure, in case of coexistence of ticketing and NFC applications. Even though 16 bytes are transferred to NTAG21x, only the least significant 4 bytes (bytes 0 to 3) are written to the specified address. Set all the remaining bytes, 04h to 0Fh, to logic 00h. Byte 0 = 0xA0 Byte 1 = Address to Write Byte 2 to 17 = 16 Bytes of Data to Write (only least significant 4 bytes are written) Note: This command is sent in 2 steps, which the Firmware will handle (1) <CMD><Address to Write><CRCH><CRCL> (2) <16 Bytes of Data to Write><CRCH><CRCL>

Command	Length	Field Value
READ_CNT	2	The READ_CNT command is used to read out the current value of the NFC one-way counter of the NTAG213, NTAG215 and NTAG216. The command has a single argument specifying the counter number and returns the 24-bit counter value of the corresponding counter. If the NFC_CNT_PWD_PROT bit is set to 1b the counter is password protected and can only be read with the READ_CNT command after a previous valid password authentication Byte 0 = 0x39 Byte 1 = 0x02 (NFC Counter Address)
PWD_AUTH	5	A protected memory area can be accessed only after a successful password verification using the PWD_AUTH command. The AUTH0 configuration byte defines the protected area. It specifies the first page that the password mechanism protects. The level of protection can be configured using the PROT bit either for write protection or read/write protection. The PWD_AUTH command takes the password as parameter and, if successful, returns the password authentication acknowledge, PACK. By setting the AUTHLIM configuration bits to a value larger than 000b, the number of unsuccessful password verifications can be limited. Each unsuccessful authentication is then counted in a counter featuring anti-tearing support. After reaching the limit of unsuccessful attempts, the memory access specified in PROT, is no longer possible. Byte 0 = 0x1B Byte 1..4 = password (4 bytes)
READ_SIG	2	The READ_SIG command returns an IC specific, 32-byte ECC signature, to verify NXP Semiconductors as the silicon vendor. The signature is programmed at chip production and cannot be changed afterwards. Byte 0 = 0x3C Byte 1 = 0x00, RFU

Response Data For NFC Tag Type 2

Tag	Len	Value / Description
81	01	Tag Response Code 0x00 = Success 0x01 = Failed
82	var	Encryption Control Payload If unencrypted: Tag / Len / Value / Description FC / var / NFC Data Container /DF7A / var / NFC Data If encrypted: /DFDF59 / var / Encrypted Data Primitive to be decrypted. /DFDF50 / var / Encrypted Data KSN /DFDF51 / 01 / Encrypted Data Encryption Type

Example Unencrypted Payload for Fast Read

```
81 0100 (Tag Respons Code)
82 82036D (Encryption Control)
   FC 820369 (NFC Data Container)
      DF7A 820364 (NFC Data)
           031391010F55047777772E6D616774656B2E636F6DFE00. . .
                                www.magtek.com
```

Example Encrypted Payload for Fast Read

```
81 0100 (Tag Response code)
82 820389 (Encryption Control)
   DFDF59 820370 (Encrypted Data Primitive)
          03679DC03B4CA607E3A7D2B52C8E9F1B5CD3D85E7368425. . .
   DFDF50 0A (Encrypted Data KSN)
          FFFF9876543210200047
   DFDF51 01 (Encrypted Data Type)
          80
```

4.18 sendClassicNFCCommand

This function sends a command to a Mifare Classic NFC tag type 2. The NFC tag must first be activated by calling startTransaction() with NFC enabled.

```
bool IDevice-> sendClassicNFCCommand (
      IData data,
      bool lastCommand,
      bool encrypt);
```

Parameter	Description
data	Command to send to the NFC tag. • `Get Version` • `Read` • `Fast Read` • `Write` • `Compatibility Write` • `Read Cnt` • `PWD Auth` • `Read Sig` For details of the command see NFC commands table below or ***D998200383 DynaFlex Family Programmer's Manual (COMMANDS) section NFC/Mifare Pass Through Commands***
lastCommand	Determines if this is the last NFC command to complete the operation. `true` = This is the last command. Device will provide a single beep after receiving a successful response from the NFC tag. To send subsequent commands, the NFC tag must be activated by calling startTransaction() with NFC enabled. `false` = Expect more commands (Default). Either set to true or false, if the NFC tag command fails, device will provide a double beep.
encrypt	Determines if data returned is to be encrypted. `true` = Encrypt data `false` = Do not encrypt data (Default)

Return Value: True of successful. Otherwise, returns false.

NFC Commands

Command	Length	Field Value
Get Version	1	The GET_VERSION command is used to retrieve information on the NTAG family, the product version, storage size and other product data required to identify the specific NTAG21x. Byte 0 = 0x60

Command	Length	Field Value
Read	2	The READ command requires a start page address, and returns the 16 bytes of four NTAG21x pages. For example, if address is 03h then pages 03h, 04h, 05h, 06h are returned. Special conditions apply if the READ command address is near the end of the accessible memory area. The special conditions also apply if at least part of the addressed pages is within a password protected area. Byte 0 = 0x30 Byte 1 = Start Page Address
Fast Read	3	The FAST_READ command requires a start page address and an end page address and returns the all n*4 bytes of the addressed pages. For example, if the start address is 03h and the end address is 07h then pages 03h, 04h, 05h, 06h and 07h are returned. Byte 0 = 0x3A Byte 2 = Start Page Address Byte 3 = End Page Address
Write	6	The WRITE command requires a block address, and writes 4 bytes of data into the addressed NTAG21x page. Byte 0 = 0xA2 Byte 1 = Address to Write Byte 2 to 5 = 4 Bytes of Data to Write
Compatibility Write	18	The COMPATIBILITY_WRITE command is implemented to guarantee interoperability with the established MIFARE Classic PCD infrastructure, in case of coexistence of ticketing and NFC applications. Even though 16 bytes are transferred to NTAG21x, only the least significant 4 bytes (bytes 0 to 3) are written to the specified address. Set all the remaining bytes, 04h to 0Fh, to logic 00h. Byte 0 = 0xA0 Byte 1 = Address to Write Byte 2 to 17 = 16 Bytes of Data to Write (only least significant 4 bytes are written) Note: This command is sent in 2 steps, which the Firmware will handle (1) <CMD><Address to Write><CRCH><CRCL> (2) <16 Bytes of Data to Write><CRCH><CRCL>

Command	Length	Field Value
READ_CNT	2	The READ_CNT command is used to read out the current value of the NFC one-way counter of the NTAG213, NTAG215 and NTAG216. The command has a single argument specifying the counter number and returns the 24-bit counter value of the corresponding counter. If the NFC_CNT_PWD_PROT bit is set to 1b the counter is password protected and can only be read with the READ_CNT command after a previous valid password authentication Byte 0 = 0x39 Byte 1 = 0x02 (NFC Counter Address)
PWD_AUTH	5	A protected memory area can be accessed only after a successful password verification using the PWD_AUTH command. The AUTH0 configuration byte defines the protected area. It specifies the first page that the password mechanism protects. The level of protection can be configured using the PROT bit either for write protection or read/write protection. The PWD_AUTH command takes the password as parameter and, if successful, returns the password authentication acknowledge, PACK. By setting the AUTHLIM configuration bits to a value larger than 000b, the number of unsuccessful password verifications can be limited. Each unsuccessful authentication is then counted in a counter featuring anti-tearing support. After reaching the limit of unsuccessful attempts, the memory access specified in PROT, is no longer possible. Byte 0 = 0x1B Byte 1..4 = password (4 bytes)
READ_SIG	2	The READ_SIG command returns an IC specific, 32-byte ECC signature, to verify NXP Semiconductors as the silicon vendor. The signature is programmed at chip production and cannot be changed afterwards. Byte 0 = 0x3C Byte 1 = 0x00, RFU

5 DeviceCapability

Create an instance of the **DeviceCapability** using **IDevice** getDeviceCapabilities**()**. Then use the functions described in this chapter.

5.1 BatteryBackedClock

This property returns true if the device is equipped with a battery that preserves the internal clock when not powered by a host system or charging.

```
boolean DeviceCapability.BatteryBackedClock;
```

Return Value: True if device is equipped with a battery backed clock. Otherwise, returns false.

5.2 Display

This property returns true if the device is equipped with display.

```
boolean DeviceCapability.Display;
```

Return Value: True if device is equipped with a display. Otherwise, returns false.

5.3 MSRPowerSaver

This property returns true if the device has the option to disable or enable the magnetic stripe reader head (MSR). The MSR may be powered down while the device is idle to minimize power consumption.

```
boolean DeviceCapability.MSRPowerSaver;
```

Return Value: True if device supports MSR power saver. Otherwise, returns false.

5.4 PaymentMethods

This property returns an integer representing **PaymentMethods** supported by the device.

```
int DeviceCapability.PaymentMethods;
```

Return Value: Int for payment method enumeration.

5.5 PINPad

This property returns true if the device is equipped with a PIN Pad.

```
bool DeviceCapability.PINPad;
```

Return Value: True if device is equipped with a PIN Pad. Otherwise, returns false.

5.6 Signature

This property returns true if the device is equipped with signature capture.

```
bool DeviceCapability.Signature;
```

Return Value: True if device is equipped with signature capture. Otherwise, returns false.

5.7 SRED

This property returns true if the device supports Secure Reading and Exchange of Data.

```
bool DeviceCapability.SRED;
```

Return Value: True if device supports SRED. Otherwise, returns false.

5.8 AutoSignatureCapture

This property returns true if the device supports auto signature capturing.

```
bool DeviceCapability.AutoSignatureCapture;
```

Return Value: True if device supports SRED. Otherwise, returns false.

6 IDeviceControl

Create an instance of the **IDeviceControl** using **IDevice getDeviceControl**. Then use the function calls described in this chapter.

Generally, these functions will run in one of two modes:

- **Asynchronous** functions return data in the event handlers in section Error! Reference source not found..
- **Synchronous** functions return data in the return value. If the data is not available immediately, the call will block until a wait time has elapsed.

6.1 close

This function closes the connection to the device.

```
bool IDeviceControl->close();
```

Return Value: True if successful. Otherwise, returns false.

6.2 deviceReset

This function resets the device. This is equivalent to a power reset. After the reset, connection to the device will need to be re-established.

```
bool IDeviceControl->deviceReset();
```

Return Value: True if successful. Otherwise, returns false.

6.3 displayMessage

This function displays a predefined message on the device.

```
bool IDeviceControl->displayMessage(
      unsigned char messageID,
      unsigned char timeout);
```

Parameter	Description
messageID	Value representing the message. Usage: 0x00 - reserved, do not use. 0x01 - “AMOUNT” 0x02 - “AMOUNT OK?” 0x03 - “APPROVED” 0x04 - “CALL YOUR BANK” 0x05 - “CANCEL OR ENTER” 0x06 - “CARD ERROR” 0x07 - “DECLINED” 0x08 - “ENTER AMOUNT” 0x09 - reserved, do not use. 0x0A - reserved, do not use. 0x0B - “INSERT CARD” 0x0C – “NOT ACCEPTED” 0x0D - reserved, do not use. 0x0E - “PLEASE WAIT” 0x0F - “PROCESSING ERROR” 0x10 - “REMOVE CARD” 0x11 - “USE CHIP READER” 0x12 - “USE MAGSTRIPE” 0x13 - “TRY AGAIN” 0x14 - “WELCOME” 0x15 - “PRESENT CARD” 0x16 - “PROCESSING” 0x17 - “CARD READ OK - REMOVE CARD” 0x18 - “INSERT OR SWIPE CARD” 0x19 - “PRESENT ONE CARD ONLY” 0x1A - “APPROVED PLEASE SIGN” 0x1B - “AUTHORIZING PLEASE WAIT” 0x1C - “INSERT, SWIPE OR TRY ANOTHER CARD” 0x1D - “PLEASE INSERT CARD” 0x1E - Null prompt (empty screen) 0x1F - reserved, do not use. 0x20 - “SEE PHONE” 0x21 - “PRESENT CARD AGAIN” 0x22 - “INSERT/SWIPE/TRY OTHER CARD” 0x23 - “TAP or SWIPE CARD” 0x24 - “TAP or INSERT CARD” 0x25 - “TAP, INSERT or SWIPE CARD” 0x26 - “TAP CARD” 0x27 - “TIMEOUT” 0x28 - “TRANSACTION TERMINATED”

Parameter	Description
timeout	Timeout in seconds for the device to display the message. Usage: `0x00` - Infinite timeout. Device leaves the requested message on the display until the host initiates a change. All other values - Timeout in seconds for the device to display the message.

Return Value: True if successful. Otherwise, returns false.

6.4 endSession

This function clears session data and returns the device to an idle state.

```
bool IDeviceControl->endSession();
```

Return Value: True if successful. Otherwise, returns false.

6.5 getInput

This function sends a request for user input at the device. The response data will be returned in the event Error! Reference source not found..

```
bool IDeviceControl->getInput(IData data);
```

Parameter	Description
data	Byte array or string data to send to the device.

Return Value: True if successful. Otherwise, returns false.

6.6 open

This function opens a connection to the device.

```
bool IDeviceControl->open();
```

Return Value: True if successful. Otherwise, returns false.

6.7 playSound

This function instructs the device to play a tone.

`bool IDeviceControl->playSound(`**IData** `data);`

Parameter	Description
data	Byte array or string data to send to the device.

Return Value: True if successful. Otherwise, returns false.

6.8 send

This function sends a command to the device. The response will be passed to the event OnEvent().

`bool IDeviceControl->send(`**IData** `data);`

Parameter	Description
data	Byte array or string data to send to the device. Data must contain the full command as required by the device.

Return Value: True if successful. Otherwise, returns false.

6.9 sendExtendedCommand

This function sends an extended command to the device. The response will be passed to the event **OnEvent()**.

`bool IDeviceControl->sendExtendedCommand(`**IData** `data);`

Parameter	Description
data	Byte array or string data to send to the device. Data must contain the full command as required by the device.

Return Value: True if successful. Otherwise, returns false.

6.10 sendSync

This function sends a synchronous command to the device. The response from the device will be returned in IResult.

`IResult IDeviceControl->sendSync(`**IData** `data);`

Parameter	Description
data	Byte array or string data to send to the device.

Return Value: `IResult`.

```
public interface IResult
{
  StatusCode Status
  IData Data
}
```

6.11 setDateTime

This function sets the date and time for the device.

`bool IDeviceControl->setDateTime(`**IData** `data);`

Parameter	Description
data	Byte array or string data to send to the device.

Return Value: True if successful. Otherwise, returns false.

6.12 setLatch

This function send a command to lock or unlock the card latch. The host can choose to lock the card during EMV transactions to limit the possibility of the cardholder prematurely removing the card. The lock can also be enabled while the card is out of the system to block cardholders from inserting a card.

```
bool IDeviceControl->setLatch(bool enableLock);
```

Parameter	Description
enableLock	Usage: `false` - unlock the latch in the device. `true` - lock the latch in the device.

Return Value: True if successful. Otherwise, returns false.

6.13 showBarCode

This function sends a command to show a barcode on the device's display.

```
bool IDeviceControl->showBarCode(
      BarCodeRequest data
      unsigned char timeout
      IData prompt = NULL);
```

Parameter	Description
data	BarCodeRequest object containing the barcode data to display.
timeout	Display Time. Usage: 0x00 = Indefinite 0x01 to 0xFF = 1 to 255 seconds
prompt	Text to display below the QR code. In Landscape orientation, the limit is approximately 30 characters. In Portrait orientation, the limit is approximately 22 characters.

`BarCodeRequest:`

Parameter	Type	Description
Type	BarCodeType	Enum to specify the type of barcode
Format	BarCodeFormat	Enum to specify the barcode format
Data	vector<unsigned char>	Data to encode into a barcode
BlockColor	vector<unsigned char>	Block color. Use RRGGBB format. 0x000000 = Black
BackgroundColor	vector<unsigned char>	Background color. Use RRGGBB format. 0xFFFFFF = White

Parameter	Type	Description
ErrorCorrection	unsigned char	Error Correction 0x00 = Low (default) 0x01 = Medium 0x02 = Quartile 0x03 = High See **ISO/IEC 18004:2015**
MaskPattern	unsigned char	Mask Pattern 0x00 to 0x07 = Mask Pattern 0xFF = Device Select Optimal Mask Pattern (default) See **ISO/IEC 18004:2015**
MinVersion	unsigned char	Minimum Version. Must be less than or equal to Maximum Version. 0x01 to 0x28 = Version 1 to Version 40 (0x01 is default) See **ISO/IEC 18004:2015**
MaxVersion	unsigned char	Maximum Version. Must be greater than or equal to Minimum Version. 0x01 to 0x28 = Version 1 to Version 40 (0x28 is default) See **ISO/IEC 18004:2015**

Return Value: True if successful. Otherwise, returns false.

6.14 showImage

This function sends a command to immediately show an image on the device's display. The image must already be loaded into a slot.

```
bool IDeviceControl->showImage(unsigned char imageID);
```

Parameter	Description
imageID	Usage: `0x01` – show the image at slot 1. `0x02` – show the image at slot 2. `0x03` – show the image at slot 3. `0x04` – show the image at slot 4.

Return Value: True if successful. Otherwise, returns false.

6.15 showImage

This function sends a command to immediately upload and show an image on the device's display.

```
bool IDeviceControl->showImage(
      ImageData data
      unsigned char timeout);
```

Parameter	Description
data	ImageData object containing the image to display.

Parameter	Description
timeout	Display Time Usage: 0x00 = Indefinite 0x01 to 0xFF = 1 to 255 seconds

`ImageData`

Member	Type/ Format	Description
type	ImageType	Enum for image type. Usage: IMAGE_TYPE_BITMAP = BMP file
data	vector<unsigned char>	Image encoded data.
backgroundColor	vector<unsigned char>	Background color in RRGGBB format. 0x000000 = Black 0xFFFFFF = White

Return Value: True if successful. Otherwise, returns false.

6.16 startBarCodeReader

This function sends a command to start the barcode reader.

```
bool IDeviceControl->startBarCodeReader(
      unsigned char timeout
      unsigned char encryptionMode);
```

Parameter	Description
timeout	Time to enable barcode reader. Usage: 0x00 = Wait until a barcode is read or **stopBarCodeReader()** is called. 0x01 to 0xFF = 1 to 255 seconds
encryptionMode	Encrypt payload Usage: 0x00 = do not encrypt barcode data 0x01 = encrypt barcode data.

Return Value: True if successful. Otherwise, returns false.

6.17 stopBarCodeReader

This function sends a command to stop the barcode reader. This is applicable only when the timeout value for startBarCodeReader() was set to 0x00.

```
bool IDeviceControl->stopBarCodeReader();
```

Returns Value: True if successful. Otherwise, returns false.

7 ConnectionInfo

Create an instance of the **ConnectionInfo** using **IDevice getConnectionInfo()**. Then use the function calls described in this chapter.

7.1 Address

This property returns the address of the device.

```
string ConnectionInfo.Address();
```

Return Value: The address of the device.

7.2 ConnectionType

This property returns the type of connection Interface for the device.

```
MTU_DEVICE_CONNECTION_TYPE ConnectionInfo.ConnectionType();
```

Return Value: `MTU_DEVICE_CONNECTION_TYPE`

7.3 DeviceType

This property returns the type for the device.

```
MTU_DEVICE_TYPE ConnectionInfo.DeviceType();
```

Return Value: `MTU_DEVICE_TYPE`

7.4 CertificateInfo

This property returns the type certification information.

```
CertificateInfo ConnectionInfo.CertificateInfo();
```

Return Value: `CertificateInfo.`

8 DeviceInfo

Create an instance of the **DeviceInfo** from **IDevice getDeviceInfo()**. Then use the function calls described in this chapter.

8.1 Model

This property returns the model name of the device.

```
string DeviceInfo.Model();
```

Return Value: The model name of the device.

8.2 Name

This property returns the name of the device.

```
string DeviceInfo.Name();
```

Return Value: The name of the device.

8.3 Serial

This property returns the serial number of the device.

```
string DeviceInfo.Serial();
```

Return Value: The serial number of the device.

9 IDeviceConfiguration

Create an instance of the **IDeviceConfiguration** using IDevice getConfigeration(). Then use the function calls described in this chapter.

Generally, these functions will run in one of two modes:

- **Asynchronous** functions return data in the event handlers in section **IConfigurationCallback Delegates**.
- **Synchronous** functions return data in the return value. If the data is not available immediately, the call will block until a wait time has elapsed.

9.1 getChallengeToken

This function retrieves a challenge token from the device. A challenge token consists of a random nonce or timestamp. A challenge token must be used within the time allowed by the device (generally 5 minutes) of being issued. Only one token can be active at a time. Attempts to use a token for requests other than the one specified will cause the token to be revoked/erased.

```
vector<unsigned char> IDeviceConfiguration->getChallengeToken(
      vector<unsigned char> data);
```

Parameter	Description
data	Array containing the request ID to be protected.

Return Value: An array containing the challenge token.

9.2 getConfigInfo

This function retrieves device configuration information. For an example, see appendix IDeviceConfiguration Walk Through.

```
vector<unsigned char> IDeviceConfiguration->getConfigInfo(
      unsigned char configType,
      vector<unsigned char> data);
```

Parameter	Description
configType	Type of configuration. For DynaFlex, this is the first number of the Property OID.
data	Configuration data to be sent to the device. For DynaFlex, this is the remainder of the constructed OID. For constructing the OID see ***D998200383 DynaFlex Family Programmer's Manual (COMMANDS)***

Return Value: An array containing the configuration information.

9.3 getDeviceInfo

This function retrieves device specific information.

```
string IDeviceConfiguration->getDeviceInfo(InfoType infoType);
```

Parameter	Description
infoType	Enumerated information type.

Return Value: A string value for device information.

9.4 getKeyInfo

This function retrieves key information.

```
vector<unsigned char> IDeviceConfiguration->getKeyInfo(
      unsigned char keyType,
      vector<unsigned char> data);
```

Parameter	Description
keyType	Type of key. For DynaFlex, use 0.
data	Key data to be sent to the device. For DynaFlex, this is the 2-byte key slot number.

Return Value: An array containing the key information.

9.5 getFile

This function sets device configuration information.

```
int IDeviceConfiguration->getFile(
      vector<unsigned char> fileID,
      IConfigurationCallback* callback);
```

Parameter	Description
fileID	Byte array for the file ID. For DynaFlex, use a 4-byte file id.
callback	Name of a class that implements the IConfigurationCallback Delegates events.

Return Value: 0 if the asynchronous configuration operation started. Otherwise, returns a non 0 value.

9.6 getKeyInfo

This function retrieves key information.

```
vector<unsigned char> IDeviceConfiguration->getKeyInfo(
      unsigned char keyType,
      vector<unsigned char> data);
```

Parameter	Description
keyType	Type of key. For DynaFlex, use 0.
data	Key data to be sent to the device. For DynaFlex, this is the 2-byte key slot number.

Return Value: An array containing the key information.

9.7 sendFile

This function sends a file to the device.

```
int IDeviceConfiguration->sendFile(
      vector<unsigned char> fileType,
      vector<unsigned char> data,
      IConfigurationCallback* callback);
```

Parameter	Description
fileID	Byte array for the file ID. For DynaFlex, use a 4-byte file id.
data	File contents to be sent to the device.
callback	Name of a class that implements the IConfigurationCallback events.

Return Value: 0 if the asynchronous update operation started. Otherwise, returns a non 0 value.

9.8 sendImage

This function sends an image to the device.

```
int IDeviceConfiguration->sendImage(
      unsigned char imageID,
      vector<unsigned char> data,
      IConfigurationCallback* callback);
```

Parameter	Description
imageID	Value for the image ID. For DynaFlex, use: `1, 2, 3, or 4`
data	File contents to be sent to the device.
callback	Name of a class that implements the IConfigurationCallback events.

Return Value: 0 if the asynchronous update operation started. Otherwise, returns a non 0 value.

9.9 setConfigInfo

This function sets device configuration information. For an example, see **API Walk Through.**

```
int IDeviceConfiguration->setConfigInfo(
      unsigned char configType,
      vector<unsigned char> data,
      IConfigurationCallback* callback);
```

Parameter	Description
configType	Type of configuration. For DynaFlex, this is the first number of the Property OID.
data	Configuration data to be sent to the device. For DynaFlex, this is the remainder of the constructed OID and value. For constructing the OID see ***D998200383 DynaFlex Family Programmer's Manual (COMMANDS)***
callback	Name of a class that implements the IConfigurationCallback events.

Return Value: 0 if the asynchronous configuration operation started. Otherwise, returns a non 0 value.

9.10 setDisplayImage

This function sets which image is to be displayed when the device is idle. The image is displayed after a device reset.

```
int IDeviceConfiguration->setDiaplayImge(unsigned char imageID);
```

Parameter	Description
imageID	Value for the image ID. For DynaFlex, use: `0, 1, 2, 3, or 4` `0` restores the image to the "Welcome" screen.

Return Value: 0 if the asynchronous configuration operation started. Otherwise, returns a non 0 value.

9.11 updateFirmware

This function updates the device firmware.

```
int IDeviceConfiguration->updateFirmware(
      unsigned short firmwareType,
      vector<unsigned char> data,
      IConfigurationCallback* callback);
```

Parameter	Description
firmwareType	Type of firmware. For DynaFlex, use: `1` - Main App `2` - Wireless App
data	Firmware image to be sent to the device.
callback	Name of a class that implements the IConfigurationCallback events. **OnProgress()** event provides in succession the progress from 0 to 100 percent during the update. **OnResult()** event provides a **StatusCode** once completed.

Return Value: 0 if the asynchronous update operation started. Otherwise, returns a non 0 value.

9.12 updateKeyInfo

This function updates key information in the device.

```
int updateKeyInfo(
      unsigned char keyType,
      vector<unsigned char> data,
      IConfigurationCallback* callback);
```

Parameter	Description
keyType	Type of key.

Parameter	Description
data	Key data to be sent to the device.
callback	Name of a class that implements the IConfigurationCallback events.

Return Value: 0 if the asynchronous update operation started. Otherwise, returns a non 0 value.

10 Data Classes

10.1 BarCodeData

BarCodeData		
Member	**Type/ Format**	**Description**
Data	vector<unsigned char>	Data payload
Encrypted	bool	0x00 = data is not encrypted 0x01 = data is encrypted
EncryptionType	unsigned char	Encryption type
KSN	vector<unsigned char>	Key Serial Number

10.2 BarCodeRequest

BarCodeRequest		
Member	**Type/ Format**	**Description**
Type	BarCodeType	Enum to specify the type of barcode
Format	BarCodeFormat	Enum to specify the barcode format
Data	vector<unsigned char>	Data to encode into a barcode
BlockColor	vector<unsigned char>	Block color. Use RRGGBB format. 0x000000 = Black
BackgroundColor	vector<unsigned char>	Background color. Use RRGGBB format. 0xFFFFFF = White
ErrorCorrection	unsigned char	Error Correction 0x00 = Low (default) 0x01 = Medium 0x02 = Quartile 0x03 = High See **ISO/IEC 18004:2015**
MaskPattern	unsigned char	Mask Pattern 0x00 to 0x07 = Mask Pattern 0xFF = Device Select Optimal Mask Pattern (default) See **ISO/IEC 18004:2015**
MinVersion	unsigned char	Minimum Version. Must be less than or equal to Maximum Version. 0x01 to 0x28 = Version 1 to Version 40 (0x01 is default) See **ISO/IEC 18004:2015**

BarCodeRequest		
MaxVersion	unsigned char	Maximum Version. Must be greater than or equal to Minimum Version. 0x01 to 0x28 = Version 1 to Version 40 (0x28 is default) See **ISO/IEC 18004:2015**

10.3 CertificateInfo

CertificateInfo is used for the connection to a devices requiring client credentials. See the following for details on installing a certificate chain: ***D998200550 DYNAFLEX TLS CERTIFICATE INSTALLATION MANUAL***

CertificateInfo		
Member	**Type/ Format**	**Description**
format	string	Certificate data format. `"PKCS12"` – for .p12 file. `"PFX"` – for .pfx file.
Data	vector<unsigned char>	Certificate data
Password	string	Password to access the certificate data.

10.4 IData

IData is used for the payload of events and passing data to functions. When assigning the member StringValue, the member ByteArray is automatically assigned. Same is true vice versa. In this way either a string or an array can be accessed without need of data conversion.

IData		
Member	**Type/ Format**	**Description**
StringValue	string	String value
ByteArray	vector<unsigned char>	Byte array

10.5 MTUSDK_ITransaction

This is the interface used as the parameter for startTransaction(). For an example, see **API Walk Through**.

ITransaction		
Member	**Type/ Format**	**Description**
PaymentMethods	int	List of the PaymentMethod enumeration. Usage: `PAYMENT_METHOD_MSR` - For magnetic stripe cards. `PAYMENT_METHOD_Contact` - For EMV chip cards. `PAYMENT_METHOD_Contactless` - For NFC contactless cards. `PAYMENT_METHOD_ManualEntry` - For user to manually enter transaction data without any card access. When set to ManualEntry, the other payment methods do not apply. `PAYMENT_METHOD_BARCODE` - For Barcode. `PAYMENT_METHOD_BARCODE_ENCRYPTED` - For encrypted Barcode. `PAYMENT_METHOD_APPLE_VAS` - For Apple VAS. `PAYMENT_METHOD_NFC` - For NFC tag.
QuickChip	bool	In QuickChip mode, the device does not prompt for an amount. Device sends an ARQC request to the host. Device automatically populates the ARPC response data with EMV Tag 8A set to "Z3". Card holder is prompted to remove the card. Transaction result is later determined by the processor and not by the card. Usage: `false` - Do not enable QuickChip mode. `true` - Enable QuickChip mode. Default.
Timeout	unsigned char	Transaction timeout in seconds. Default is 60 seconds. Usage: `0 to 255` - Depending on the device, 0 means no timeout.
Amount	string Max 13	EMV Tag 9F02 - Authorized amount of the transaction. Example: `"1.23"` – $1.23 `"10"` – $10.00

ITransaction		
CashBack	string Max 13	EMV Tag 9F03 - Secondary amount associated with the transaction. Example: `"1.23"` – $1.23 `"10"` – $10.00
CurrenyCode	vector<unsigned char> 2	EMV Tag 5F2A - Currency code of the transaction according to ISO 4217. The byte array is null by default. Example: `0x0840` = US Dollar `0x0978` = Euro `0x0826` = UK Pound
CurrencyExponent	vector<unsigned char> 1	EMV Tag 5F36 - The decimal point position from the right of the transaction amount. The byte array is null by default. Example: `0x02` – decimal point at 2 position from the right.
EMVOnly	bool	Flag that determines whether or not to start a transaction in EMV mode. This takes effect for devices which support both MSR and EMV mode (not limited to eDynamo, tDynamo, and DynaPro Family). This has no effect on non EMV devices. Usage: `false` - Do not start transaction in EMV mode. `true` - Only start transaction in EMV mode. Default.
PreventMSRSignatureForCardWithICC	bool	Flag that forces the device to skip signature capture during an MSR-only transaction if the card's service code indicates it is a chip card. Usage: `false` - Allow the prompt for a signature if requested. `true` - Do not prompt for signature.
SuppressThankYouMessage	bool	By default, devices with a display signal the end of a transaction by briefly showing "THANK YOU," then "WELCOME." Usage: `false` - Do not suppress the thank you message. `true` - Suppress the thank you message.

OverrideFinalTransactionMessage	unsigned char	By default, devices with a display signal the end of a transaction by returning to the idle page and showing "WELCOME." This parameter directs the device to show a message based on the Message ID from the command displayMessage(). This option completely overrides the device's idle page behavior until the next transaction, power cycle, or other similar state change. Usage: 0x00 - reserved, do not use. 0x01 - "AMOUNT" 0x02 - "AMOUNT OK?" 0x03 - "APPROVED" 0x04 - "CALL YOUR BANK" 0x05 - "CANCEL OR ENTER" 0x06 - "CARD ERROR" 0x07 - "DECLINED" 0x08 - "ENTER AMOUNT" 0x09 - reserved, do not use. 0x0A - reserved, do not use. 0x0B - "INSERT CARD" 0x0C – "NOT ACCEPTED" 0x0D - reserved, do not use. 0x0E - "PLEASE WAIT" 0x0F - "PROCESSING ERROR" 0x10 - "REMOVE CARD" 0x11 - "USE CHIP READER" 0x12 - "USE MAGSTRIPE" 0x13 - "TRY AGAIN" 0x14 - "WELCOME" 0x15 - "PRESENT CARD" 0x16 - "PROCESSING" 0x17 - "CARD READ OK - REMOVE CARD" 0x18 - "INSERT OR SWIPE CARD" 0x19 - "PRESENT ONE CARD ONLY" 0x1A - "APPROVED PLEASE SIGN" 0x1B - "AUTHORIZING PLEASE WAIT" 0x1C - "INSERT, SWIPE OR TRY ANOTHER CARD" 0x1D - "PLEASE INSERT CARD" 0x1E - Null prompt (empty screen) 0x1F - reserved, do not use. 0x20 - "SEE PHONE" 0x21 - "PRESENT CARD AGAIN" 0x22 - "INSERT/SWIPE/TRY OTHER CARD" 0x23 - "TAP or SWIPE CARD" 0x24 - "TAP or INSERT CARD" 0x25 - "TAP, INSERT or SWIPE CARD" 0x26 - "TAP CARD" 0x27 - "TIMEOUT" 0x28 - "TRANSACTION TERMINATED"

ITransaction		
EMVResponseFormat	unsigned char	The format of the EMV response. Usage: 0x00 – Legacy. Default. 0x01 – RFU
MerchantCategory	vector<unsigned char> 2	EMV Tag 9F15 - The type of business being done by the merchant, represented according to ISO 18245. The byte array is null by default.
MerchantCustomData	vector<unsigned char> 20	EMV Tag 9F7C – Proprietary merchant data that may be requested. The byte array is null by default.
MerchantID	vector<unsigned char> 15	EMV Tag 9F16 - Used to uniquely identify a given merchant. The byte array is null by default.
TransactionCategory	vector<unsigned char> 1	EMV Tag 9F53 - The type of contactless transaction being performed. The byte array is null by default.
TransactionType	byte 1	EMV Tag 9C - The type of financial transaction, represented by the first two digits of the ISO 8583:1987 Processing Code. Usage: `0x00` – purchase. Default. `0x01` – cash advance `0x09` – purchase with cashback `0x20` – refund Supported transaction types can found in the commands programmers manual specific to the device.
ManualEntryType	unsigned char	User interface sequence. Usage: `0x00` - Card Number, Expiration Date, Security Code `0x01` - Name on Card, Card Number, Expiration Date, Security Code (Reserved for Future Use) `0x02` – Qwick Code, Last 4 digits of Card Number, Security Code (Reserved for future use)
ManualEntryFormat	unsigned char	Card number valid format. Usage: `0x00` - PAN min 8, max 21 digits
ManualEntrySound	unsigned char	Beeper feedback. Usage: `0x00` - On keypress sound disabled `0x01` - On keypress sound enabled

ITransaction		

AppleVASMode	VASMode	enum VASMode : int
		{
		VASMode_Single = 1,
		VASMode_Dual = 2,
		VASMode_VASOnly = 3,
		};

AppleVASProtocol	VASProtocol	enum VASProtocol : int
		{
		VASProtocol_URL = 1,
		VASProtocol_Full = 2,
		};

10.6 NFCEventBuilder

This class assist in parsing NFCEvent data.

NFCEventBuilder		
Member	**Return**	**Description**
GetValue(string data)	NFCEvent	Returns the NFCEvent enumeration.
GetString(NFCEvent value)	string	Returns a string representation of NFCEvent enumeration.

10.7 UserEventBuilder

This class assist in parsing UserEvent data.

NFCEventBuilder		
Member	**Return**	**Description**
GetValue(string data)	UserEvent	Returns the UserEvent enumeration.
GetString(NFCEvent value)	string	Returns a string representation of UserEvent enumeration.

10.8 DeviceEventBuilder

This class assist in parsing DeviceEvent data.

NFCEventBuilder		
Member	**Return**	**Description**
GetValue(string data)	DevicerEvent	Returns the UserEvent enumeration.
GetString(NFCEvent value)	string	Returns a string representation of DeviceEvent enumeration.

10.9 ConnectionStateBuilder

This class assist in parsing ConnectionState data.

NFCEventBuilder		
Member	**Return**	**Description**
GetValue(string data)	ConnectionStatue	Returns the ConnectionState enumeration.
GetString(NFCEvent value)	string	Returns a string representation of ConnectionState enumeration.

ConnectionStateBuilder

This class assist in parsing ConnectionState data.

NFCEventBuilder		
Member	**Return**	**Description**
GetValue(string data)	ConnectionStatue	Returns the ConnectionState enumeration.
GetString(NFCEvent value)	string	Returns a string representation of ConnectionState enumeration.

11 IEventSubscriber

This interface invokes a callback function to receive data and/or a detailed response. To register for the event(s), call the **subscribeAll()** function with the name of a class that implements the IEventSubscriber interface.

11.1 OnEvent

This function sets a callback function to notify when the device has an ARQC message send to the host.

```
void OnEvent(
      EventType eventType,
      IData data)
```

Parameter	Description
eventType	An enumeration for EventType. This indicates the event triggered by the device.
data	Contains the data for the event. The payload is dependent on the event type.

Return Value: None

12 IConfigurationCallback Delegates

This interface invokes a callback function to receive data and/or a detailed response. To register for the event(s), call the **subscribeAll()** function with the name of a class that implements the **IConfigurationCallback Delegates** interface.

12.1 OnCalculateMAC

This event is called when certain asynchronous **IDeviceConfiguration** operations need to have a MAC included with the request.

```
IResult OnCalculateMAC(
      unsigned char MACType,
      vector<unsigned char> data);
```

Parameter	Description
MACType	Type of Mac algorithm. For DynaFlex, use 0.
data	Contains the data of the payload to MAC.

Return Value: An IResult that contains the calculated MAC.

12.2 OnProgress

This event is called to update the host on the progress of an asynchronous IDeviceConfiguration operation. Enacted during but not limited to file sending and firmware updates.

```
void OnProgress(int progress);
```

Parameter	Description
progress	The progress of the configuration operation. Range: `0 - 100`

Return Value: None

12.3 OnResult

This event is called to update the host when an asynchronous IDeviceConfiguration operation is completed.

```
void OnResult(
      int status,
      vector<unsigned char> data);
```

Parameter	Description
status	Int representing an enumerated **StatusCode**.
data	Contains the data for the event.

Return Value: None

13 Enumerations

13.1 BarCodeFormat

This enum refers to the type of barcodes to display.

Enum	Description
BARCODE_FORMAT_BLOB	Data is binary format
BARCODE_FORMAT_COMMAND	Data is a command in binary format
BARCODE_FORMAT_BLOB_BASE64	Data is Base64 encoded format
BARCODE_FORMAT_COMMAND_BASE64	Data is a command in Base64 format

13.2 BarCodeType

This enum refers to the type of barcodes to display.

Enum	Description
QRCODE	QR code

13.3 ConnectionState

This enum refers to the readiness of the SDK to communicate with the device. This is not the physical attachment to a host system.

Enum	Description
CONNECTION_STATE_Unknown	Device is in an unknown connection state.
CONNECTION_STATE_Disconnected	Device is disconnected.
CONNECTION_STATE_Connecting	Device is in the process of connecting. The next state is to be Connected.
CONNECTION_STATE_Error	There was an error either connecting or disconnecting the device.
CONNECTION_STATE_Connected	Device is connected and ready for transacting.
CONNECTION_STATE_Disconnecting	Device is in the process of disconnecting. The next state will is to be Disconnected.

13.4 MTU_DEVICE_CONNECTION_TYPE

This enum refers to the communication interface type of MagTek reader which the SDK will control.

Enum	Description
MTU_USB	Universal Serial Bus supported devices:

Enum	Description
	• eDynamo • mDynamo • Dynamag • DynaMax • tDynamo • kDynamo • cDynamo • iDynamo 6 • DynaPro • DynaPro Go • DynaPro Mini • DynaFlex • DynaFlex Pro • DynaProx
`MTU_BLE`	Bluetooth Low Energy devices: • DynaMax
`MTU_BLE_EMV`	Bluetooth Low Energy with EMV supported devices: • eDynamo
`MTU_BLE_EMVT`	Bluetooth Low Energy with EMV supported devices: • tDynamo
`MTU_TCP`	Transmission Control Protocol supported devices: • DynaPro
`MTU_TLS12`	Transmission Control Protocol with Transport Layer Security supported devices: • DynaPro Go
`MTU_TLS12_TRUST`	Transmission Control Protocol with Transport Layer Security supported devices: • DynaPro Go
`MTU_SERIAL`	UART supported devices
`MTU_WS`	WebSocket supported devices: • DynaFlex II PED
`MTU_WS_TRUST`	WebSocket supported devices. This will TLS connection to device without requirement for name match. • DynaFlex II PED
`MTU_LIGHTNING`	Audio devices: • iDynamo 6

13.5 DataEntryType

This enum is reserved for future use.

Enum	Description
`PIN`	Request Personal Identification Number
`Signature`	Request Signature

Enum	Description
SSN	Request Social security number
ZIPCODE	Request Zip code
BirthDate	Request Birth date
ActivationCode	Request Activation code

13.6 DeviceEvent

This enum refers to a change in the device status.

Enum	Description
MTU_DEVICE_EVENT_None	No event to report.
MTU_DEVICE_EVENT_DeviceResetOccurred	A device reset had occurred.
MTU_DEVICE_EVENT_DeviceResetWillOccur	A device reset will occur soon. Host application may uses this as a warning to take appropriate actions.

13.7 DeviceFeature

This enum refers to a featured supported by the device.

Enum	Description
MTU_DEVICE_FEATURE_None	No feature
MTU_DEVICE_FEATURE_SignatureCapture	Supports signature capture
MTU_DEVICE_FEATURE_PINEntry	Supports PIN entry
MTU_DEVICE_FEATURE_PANEntry	Supports PAN entry
MTU_DEVICE_FEATURE_ShowBarCode	Supports display of a barcode
MTU_DEVICE_FEATURE_ScanBarCode	Supports scanning a barcode
MTU_DEVICE_FEATURE_DisplayMessage	Supports display of a messsage

13.8 MTU_DEVICE_TYPE

This enum refers to the type of MagTek reader which the SDK will control.

Enum	Description
SCRA	Secure Reader Authenticator devices. List includes but not limited to: • eDynamo • mDynamo • Dynamag • DynaMax • tDynamo • kDynamo • cDynamo • iDynamo 6
PPSCRA	PIN Pad Secure Reader Authenticator devices. List includes but not limited to: • DynaPro • DynaPro Go • DynaPro Mini
CMF	Common Message Structure devices. List includes but not limited to: • oDynamo
MMS	MMS class devices. (MagTek Message Scheme) List includes but not limited to: • DynaFlex • DynaFlex Pro • DynaProx

13.9 FeatureStatus

This enum refers to the status of a specific feature reported from **DeviceFeature**.

Enum	Description
MTU_FEATURE_STATUS_NoStatus	No change in status
MTU_FEATURE_STATUS_Success	Success
MTU_FEATURE_STATUS_Failed	Failed
MTU_FEATURE_STATUS_TimedOut	Timed out
MTU_FEATURE_STATUS_Cancelled	Cancelled
MTU_FEATURE_STATUS_Error	Error
MTU_FEATURE_STATUS_HardwareNA	Featured hardware not applicable for a status

13.10 ImageType

This enum refers to the type of image.

Enum	Description
IMAGE_TYPE_BITMAP	BMP file

13.11 InfoType

This enum refers to the type of specific information to retrieve from the device.

Enum	Description
INFOTYPE_DeviceSerialNumber	Device serial number.
INFOTYPE_FirmwareVersion	Firmware version of the device.
INFOTYPE_DeviceCapabilities	Capabilities of the device delimited by a comma.
INFOTYPE_Boot1Version	Boot 1 firmware version of the device.
INFOTYPE_Boot0Version	Boot 0 firmware version of the device.
INFOTYPE_FirmwareHash	Firmware hash comprised of part numbers, versions, and timestamps.
INFOTYPE_TamperStatus	Tamper status of the device. 0x00 = Not Tampered 0x01 = Tampered
INFOTYPE_OperationStatus	Operation status of the device. 0x01 = Offline 0x02 = Online
INFOTYPE_OfflineDetail	Details of why the device is offline. • Bit 0 = Tamper problem present • Bit 1 = Master Key problem present • Bit 2 = Keys and Certificates problem present • Bit 3 = Real Time Clock problem present • Bit 4 = Random Number Generator problem present • Bit 5 = Cryptography Engine problem present • Bit 6 = Magnetic Stripe Reader Hardware problem present • Bit 7 = Reserved
INFOTYPE__DeviceModel	Device model.

13.12 PaymentMethods

This enum refers to which card type the device will perform a transaction.

Enum	Description
PAYMENT_METHOD_MSR	For magnetic stripe cards.
PAYMENT_METHOD_CONTACT	For EMV chip cards.
PAYMENT_METHOD_CONTACTLESS	For NFC contactless cards.

Enum	Description
PAYMENT_METHOD_MANUALENTRY	For user to manually enter transaction data without any card access.
PAYMENT_METHOD_BARCODE	For Barcode transaction.
PAYMENT_METHOD_BARCODE_Encrypted	For encrypted Barcode transaction.
PAYMENT_METHOD_APPLE_VAS	For Apple Vas transaction.
PAYMENT_METHOD_NFC	For NFC tag transaction.

13.13 MTUSDK_TransactionStatus

This enum refers to the status of the transaction.

Enum	Description
NoStatus	Set before the start of a transaction and before a card is presented to the device.
NoTransaction	No transaction in progress.
CardSwiped	A card was swiped into the device.
CardInserted	A card was inserted into the device.
CardRemoved	A card was removed from the device.
CardDetected	A card was detected by the device.
CardCollision	A card collision was detected by the device.
TimedOut	The transaction was not completed before a timeout period.
HostCanceled	The host software sent a cancel.
TransactionCanceled	The transaction was canceled.
TransactionInProgress	The transactions is in progress.
TransactionError	There is an error during the transaction.
TransactionApproved	The transactions is approved.
TransactionDeclined	The transactions is declined.
TransactionCompleted	The transaction is completed.
TransactionFailed	The transaction failed.
TransactionNotAccepted	The transaction was not accepted by the device.
SignatureCaptureRequested	A signature capture is requested by the device.
TechnicalFallback	Due to technical reasons, the chip transaction cannot be completed by the reader.
QuickChipDeferred	Device has sent a “Z3” response code to the chip card.
DataEntered	Data has been entered on the device for a manual card entry transaction.

Enum	Description
`TryAnotherInterface`	Due to removal of the chip card or error with contactless card, the transaction cannot be completed by the reader.

13.14 EventType

This enum refers to the type of event triggered by the device.

Enum	Description
`EVENT_TYPE_Invalid`	Invalid
`EVENT_TYPE_ConnectionState`	There was a change in the connection state of the device.
`EVENT_TYPE_DeviceResponse`	Device has responded to a command.
`EVENT_TYPE_DeviceExtendedResponse`	Device has responded to an extended command.
`EVENT_TYPE_DeviceNotification`	Device has sent a notification.
`EVENT_TYPE_DeviceTransferCancelled`	Device has cancelled data transfer.
`EVENT_TYPE_CardData`	Device has sent magnetic stripe data from a card swipe.
`EVENT_TYPE_TransactionStatus`	There was a change in transaction status.
`EVENT_TYPE_DisplayMessage`	Device has a message to display for the user.
`EVENT_TYPE_InputRequest`	Device is requesting input from the user.
`EVENT_TYPE_AuthorizationRequest`	Device has sent the Authorization Request Cryptogram and associated block of EMV tags for a transaction. This block is meant to be sent to the transaction processor.
`EVENT_TYPE_TransactionResult`	Device has sent the result of the transaction.
`EVENT_TYPE_PINBlock`	Device has sent the PINBlock after the user has entered a PIN on the device.
`EVENT_TYPE_Signature`	Device has sent data which represents a signature from a user.
`EVENT_TYPE_DeviceDataFile`	Device has sent a data file.
`EVENT_TYPE_OperationStatus`	Device has sent an operation status of a command.
`EVENT_TYPE_DeviceEvent`	Device has sent change of device state.
`EVENT_TYPE_UserEvent`	Device has sent a notification related to user interaction with the device.
`EVENT_TYPE_FeatureStatus`	Device has sent data related to a feature.
`EVENT_TYPE_PINData`	Device has sent data related to a PIN.
`EVENT_TYPE_PANData`	Device has sent data related to a PAN.

Enum	Description
EVENT_TYPE_BarCodeData	Device has sent barcode data.
EVENT_TYPE_NFCEvent	Device has sent an NFC event.
EVENT_TYPE_NFCData	Device has sent data related to NFC tag.
EVENT_TYPE_NFCResponse	Device has sent an NFC response.

13.15 MTU_OPERATION_STATUS

This enum refers to the operation status of the device.

Enum	Description
MTU_OS_NoStatus	No update for the operation.
MTU_OS_Started	Device has started an operation.
MTU_OS_Warning	Device has sent a warning about the operation.
MTU_OS_Failed	Device has failed an operation.
MTU_OS_Done	Device has completed an operation.

13.16 MTUSDK_USEREVENT

This enum refers to the type of user event reported by the device. These events relate to user interaction.

Enum	Description
MTU_USER_EVENT_None	No events yet to occur.
MTU_USER_EVENT_ContactlessCardPresented	Contactless card has been presented.
MTU_USER_EVENT_ContactlessCardRemoved	Contactless card has been removed.
MTU_USER_EVENT_CardSeated	Card is seated into the chip station.
MTU_USER_EVENT_CardUnseated	Card was removed from the chip station.
MTU_USER_EVENT_CardSwiped	Magnetic stripe card was swiped.
MTU_USER_EVENT_TouchPresented	Touch screen sensor press detected.
MTU_USER_EVENT_TouchRemoved	Touch screen sensor release detected.
MTU_USER_EVENT_ BarcodeRead	Barcode read detected.
MTU_USER_EVENT_ NFCMifareUltralightPresented	Mifare Ultralight presented.
MTU_USER_EVENT_ MifareClassic1KPresented	Mifare Classic 1K presented.
MTU_USER_EVENT_ MifareClassic4KPresented	Mifare Classic 4K presented.
MTU_USER_EVENT_ NFCMifareUltralightRemoved	Mifare Ultralight removed.

Enum	Description
MTU_USER_EVENT_ MifareClassic1KRemoved	Mifare Classic 1K removed
MTU_USER_EVENT_ MifareClassic4KRemoved	Mifare Classic 4K removed

13.17 StatusCode

This enum refers to the operation status of the SDK.

Enum	Description
SUCCESS	The operation completed successfully.
TIMEOUT	The operation timed out.
ERROR	Error attempting the operation.
UNAVAILABLE	Status currently unavailable.

13.18 NFCEvent

This enum refers to NFC events.

Enum	Description
MTUSDKNFCE_None	No events yet to occur.
MTUSDKNFCE_TagRemoved	Tag removed.
MTUSDKNFCE_NFCMifareUltralight	Mifare Ultralight.
MTUSDKNFCE_MifareClassic	Mifare Classic.
MTUSDKNFCE_CommandFailed	NFC command failed.
MTUSDKNFCE_MifareClassic1K	Mifare Classic 1K.
MTUSDKNFCE_MifareClassic4K	Mifare Classic 4K.
MTUSDKNFCE_Failed	Failed.
MTUSDKNFCE_IOFailed	IO failed.
MTUSDKNFCE_AuthenticationFailed	Authentication failed.

13.19 VASMode

This enum refers to the Apple VAS mode. This controls how the Apple VAS data is returned in the transaction ARQC. For details on Apple VAS data structure returned in a transaction see ***D998200383 DynaFlex Family Programmer's Manual (COMMANDS)***.

Enum	Description
`Single`	The device reads only Apple VAS data from a tapped smartphone, or reads EMV payment data from a tapped card. When the device sends ARQC to conclude the transaction, it only includes either EMV payment data in container FC for cards, or includes VAS data in container FE for smartphones.
`Dual`	The device reads both Apple VAS data and EMV payment data from a tapped smartphone, or reads EMV payment data from a tapped card. When device sends ARQC to the host to conclude the transaction, it includes EMV payment data in container FC and includes VAS data, if available, in container FE.
`VASOnly`	The device reads only Apple VAS data from a tapped smartphone, and does not read data from a tapped card. If the tapped smartphone does not support VAS, the device does not detect or read from the smartphone. When the device send ARQC to conclude the transaction, it includes VAS data in container FE and does not include EMV payment data in container FC.

13.20 VASProtocol

This enum refers to the Apple VAS protocol. For details on Apple VAS data structure returned in a transaction see ***D998200383 DynaFlex Family Programmer's Manual (COMMANDS)***.

Enum	Description
`URL`	URL VAS protocol
`Full`	Full VAS protocol

Appendix A API Walk Through

A.1 CoreAPI Walk Trough

The following walks through how to create instances of a device.

- CoreAPI.getDeviceList → IDevice

These examples demonstrate methods for creating an **IDevice** to be used in the MagTek Universal SDK.

Here, a list of **IDevice** is established. The first device is accessed at index 0.

```
#include "MTUSDK.h"
using namespace MTUSDK;

// Acess MMS with Universal SDK using getDeviceList()

vector<IDevice*> devices =
CoreAPI::getDeviceList(MTU_DEVICE_TYPE::MMS);

IDevice* mtmms = devices.at(0);
mtmms->requestSignature(0xff);
```

A.2 IDevice Walk Through

The following walks through how to make use of **IDevice**.

- Implement device events within the class to receive events.
- CoreAPI → IDevice.
- IDevice → subscribeAll().
- IDevice → other functions.
- IDevice → startTransaction().

Example

```
#include "MTUSDK.h"
using namespace MTUSDK;


// Extend the main window to receive events.
class MainWindow : public IEventSubscriber, IConfigurationCallback
{

// Establish a device from CoreAPI.
vector<IDevice*> deviceList = CoreAPI::getDeviceList();
IDevice* device = deviceList.at(0);


/* Suscribe to events sent from the device.
   These would be but not limited to: card inserted, card removed,
   connection state...

   Set MainWindow to receive the events. */
bool return = device_subscribeAll(&MainWindow);


// Assign parameters for the transaction.
MTU_ITransaction transaction;
transaction.Amount = “1.00”;
transaction.CashBack = “0.00”;
transaction.EMVOnly = true;
int PaymentMethodes = = PAYMENT_METHOD_MSR | PAYMENT_METHOD_CONTACT |
PAYMENT_METHOD_CONTACTLESS;
transaction.PaymentMethods = PaymentMethods;
transaction.QuickChip = true;

// Start transaction.
return = device->startTransaction(transaction);
```

A.2.1 Handling Events

Application Main window may extent the Error! Reference source not found., or events can be interfaced by a separate class. This example shows how to parse for various event types.

Example

```
// A class to handle events.
class MainWindow : public IEventSubscriber
{
  virtual void OnEvent(EventType eventType, IData data)
  {
    switch (eventType)
    {
```

Various events are separately shown below.

```
case EVENT_TYPE_ConnectionState:
// Parse for the ConnectionState
ConnectionState value =
ConnectionStateBuilder::GetValue(data.StringValue);

break;
```

```
case EVENT_TYPE_DeviceResponse:

break;
```

```
case EVENT_TYPE_DeviceExtendedResponse:

break;
```

```
case EVENT_TYPE_DeviceNotification:

break;
```

```
case EVENT_TYPE_CardData:

break;
```

```
case EVENT_TYPE_TransactionStatus:
// Parse for the transaction status code and detail.
MTUSDK_TransactionStatus status =
MTUSDK_TransactionStatusBuilder::GetStatusCode(data.StringValue);

string statusDetail =
MTUSDK_TransactionStatusBuilder::GetString(data.StringValue);

break;
```

```
case EVENT_TYPE_DisplayMessage:

string message;
```

```
// Get the message.
message = data.StringValue;

break;
```

```
case EVENT_TYPE_AuthorizationRequest:

// Forward the ARQC to processor.
/* data.ByteArray[0..1] – ARQC length
   data.ByteArray[2..n] – remainder contains the ARQC TLV object
*/

IData ARQC = data;

// App function to send the request to the processor.
string ARPC = sendARQCToProcessorForApproval(ARQC.ByteArray);

// Send authorization to device when not in QuickChip mode.
IData ARPCTLV;
ARPCTLV.StringValue = "FF7413DFDF250742363243413546FA067004” + ARPC;

if (transaction.QuickChip == false)
{
  device->sendAuthorization(ARPCTLV);
}

break;
```

```
case EventType.TransactionResult:

/* data.ByteArray[0]    – Signature Required
   data.ByteArray[1..2] – Batch Data length
   data.ByteArray[3..n] – remainder contains the Batch Data TLV object
*/

// Parse the TLV from data[].
// Abstract Approval status from TLV tag “DFDF1A”.
// Abstract Signature Required status from TLV tag at data.ByteArray
// [0].

break;
```

```
case EventType.TransactionStatus:

/*
Transaction status enumeration is build from
the TransactionStatusBuilder.

*/
```

```
MTUSDK_TransactionStatus status =
MTUSDK_TransactionStatusBuilder::GetStatusCode(data.StringValue);

if (status == MTUSDK_TransactionStatusBuilder::MTUSDKTS_CARD_SWIPED)
{
  //
}
if (status == MTUSDK_TransactionStatusBuilder::MTUSDKTS_CARDINSERTED)
{
  //
}

if (status == MTUSDK_TransactionStatusBuilder::
MTUSDKTS_TRANSACTION_APPROVED)
{
  //
}

break;
```

A.3 IDeviceControl Walk Through

The following walks through how to make use of **IDeviceControl**.

- IDevice → IDeviceControl.
- IDeviceControl → open().
- IDeviceControl → other functions.
- IDeviceControl → close().

Example

```
// Establish a device from CoreAPI.
vector<IDevice*> devices =
CoreAPI::getDeviceList(MTU_DEVICE_TYPE::MMS);
IDevice* device = devices.at(0);


// Establish a deviceControl from device.
IDeviceControl* deviceControl = device->getDeviceControl();

// Open the device, then use the IDeviceControl functions.
deviceControl->open();

. . .

// Close the device.
deviceControl->close();
```

A.4 ConnectionInfo Walk Through

The following walks through how to make use of **ConnectionInfo**.

- IDevice → ConnectionInfo.
- ConnectionInfo → getAddress()
- ConnectionInfo → getConnectionType()
- ConnectionInfo → getDeviceType()

Example

```
// Establish a device from CoreAPI.
vector<IDevice*> devices =
CoreAPI::getDeviceList(MTU_DEVICE_TYPE::MMS);
IDevice* device = devices.at(0);

// Establish a ConnectionInfo from device.
ConnectionInfo connectionInfo = device->getConnectionInfo();

// Retrieve address, connectionType, and deviceType.
string address = connectionInfo.getAddress();

MTU_DEVICE_CONNECTION_TYPE connectionType =
connectionInfo.getConnectionType();

MTU_DEVICE_TYPE deviceType = connectionInfo.getDeviceType();
```

A.5 DeviceCapability Walk Through

The following walks through how to make use of **DeviceCapability**

- IDevice → DeviceCapability.
- DeviceCapability → BatteryBackedClock() to check if date/time should be set.
- DeviceCapability → PaymentMethods() to check card types supported.
- DeviceCapability → other functions.

```
// Establish a device from CoreAPI.
vector<IDevice*> devices =
CoreAPI::getDeviceList(MTU_DEVICE_TYPE::MMS);
IDevice* device = devices.at(0);


// Establish a DeviceCapability from device.
DeviceCapability capabilities = device->getCapabilities();


// Retrieve device capabilities.
bool batteryBackedClock = capabilities.BatteryBackedClock();
if (batteryBackedClock)
{
  // Call IDeviceControl->setDateTime().
}

// Retrieve supported card payment methods.
PaymentMethods paymentMethods = capabilities.PaymentMethods();

. . .
```

A.6 IDeviceConfiguration Walk Through

The following walks through how to make use of **IDeviceConfiguration**.

- IDevice → getConfiguration().
- IDeviceConfiguration → updateFirmware().
- IDeviceConfiguration → getConfiguration().
- IDeviceConfiguration → setConfiguration().
- IDeviceConfiguration → other functions.

Example

```
// Establish a device from CoreAPI.
vector<IDevice*> devices =
CoreAPI::getDeviceList(MTU_DEVICE_TYPE::MMS);
IDevice* device = devices.at(0);

IDeviceConfiguration* devConfig = device->getConfiguration();
IDeviceControl* devControl = device.getDeviceControl();
devControl->open();


// Update firmware.
vector<unsigned char> data;
ifstream fw("firmware.bin", ifstream::binary);
if (fw)
{
   while (!fw.eof())
   {
      int aByte = fw.get();
      data.push_back(aByte);
   }
}

int return = devConfig->updateFirmware(0x01, data, &MainWindow);


/* Get configuration.
   Device-Driven Fallback OID = 1.2.1.1.1.1
                contructed OID = E2 08 E1 06 E1 04 E1 02 C1 00
   Note: first digit of OID is ommited in the construction and instead
is passed in the configType.
*/
unsigned char configType = 0x01;
vector<unsigned char> data2 =
{0xE2,0x08,0xE1,0x06,0xE1,0x04,0xE1,0x02,0xC1,0x00 };
vector<unsigned char> response = devConfig->getConfigInfo(configType,
data2);


/* Set configuation.
   Device-Driven Fallback OID is 1.2.1.1.1.1
```

```
      Disabled contructed OID = E2 09 E1 07 E1 05 E1 03 C1 01 00
       Enabled contructed OID = E2 09 E1 07 E1 05 E1 03 C1 01 01
   Note: first digit of OID is ommited in the construction and instead
is passed in the configType.

*/
vector<unsigned char> data3 =
{0xE2,0x09,0xE1,0x07,0xE1,0x05,0xE1,0x03,0xC1,0x01,0x00};
return = devConfig->setConfigInfo(configType, data3, &MainWindow);
```

A.6.1 Handling Events

Application Main window may extent the **IConfigurationCallback Delegates** or can be extended by a separate class. This example uses a separate class and demonstrates how to parse for the various events.

Example

```
// A class to handle configuration callback events.
class ConfigCallbacks : public IConfigurationCallback
public:
{
  virtural void OnProgress(int progress)
  {
      /* Handle progress.
         Progress is complete when progress = 100 */
  }
```

```
  virtual void OnResult(int status, vector<unsigned char> data)
  {
     /* Handle result.
        A configuration process is complete when
        status = StatusCode.Success */
  }
```

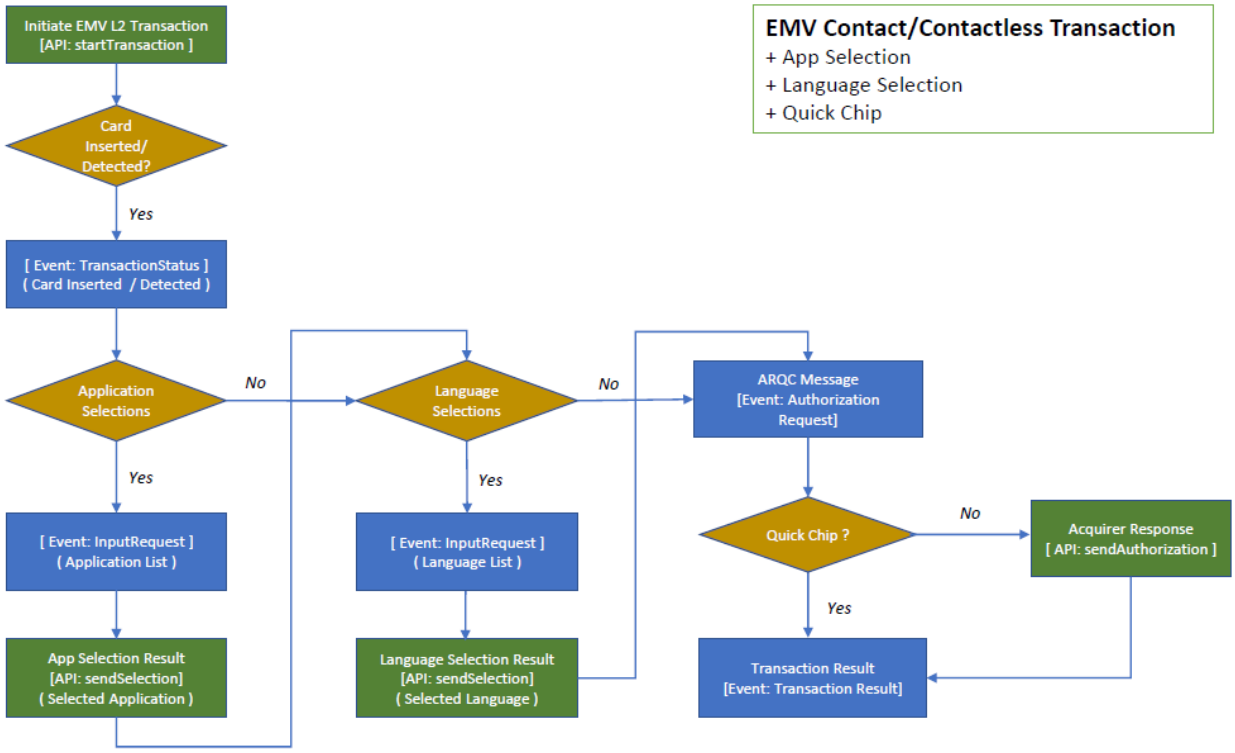
```
  IResult OnCalculateMAC(unsigned char MACType, vector<unsigned char>
data)
  {

      /* Handle MAC */
    return result;
  }
}
```

Appendix B EMV Transaction Flow

This section demonstrates transaction flows.

B.1 Flow Chart: Selections + Quick Chip



B.2 Sample Code: QuickChip

The following breaks out the EMV flow chart into code. When enabling QuickChip mode, host does not send the ARPC to the device to complete the transaction. Events are shown separately and in the order received.

```
// Assign parameters.
MTU_ITransaction transaction;
transaction.Amount = “1.00”;
transaction.CashBack = “0.00”;
transaction.EMVOnly = true;
int PaymentMethodes = = PAYMENT_METHOD_MSR | PAYMENT_METHOD_CONTACT |
PAYMENT_METHOD_CONTACTLESS;
transaction.PaymentMethods = PaymentMethods;
transaction.QuickChip = true;
transaction.QuickChip = true;  // QuickChip mode enabled.

// Start transaction.
bool result = device->startTransaction(transaction);
```

```
virtual void OnEvent(EventType eventType, IData data)
{
  string message
  switch (eventType);
    {
      case EVENT_TYPE_DisplayMessasge:
        // Get the message.
        message = data.StringValue; // “Present Card”
  }
}
```

```
virtual void OnEvent(EventType eventType, IData data)
{
  vector<unsigned car> ARQC;
  switch (eventType);
    {
      case EVENT_TYPE_AuthorizationRequest:
        // Forward ARQC to processor.
        /* data.ByteArray[0..1] – ARQC length
           data.ByteArray [2..n] – remainder contains the ARQC TLV
           object */
    }
}
```

```
public void OnEvent(EventType eventType, IData data)
{
  string message;
  switch (eventType);
    {
```

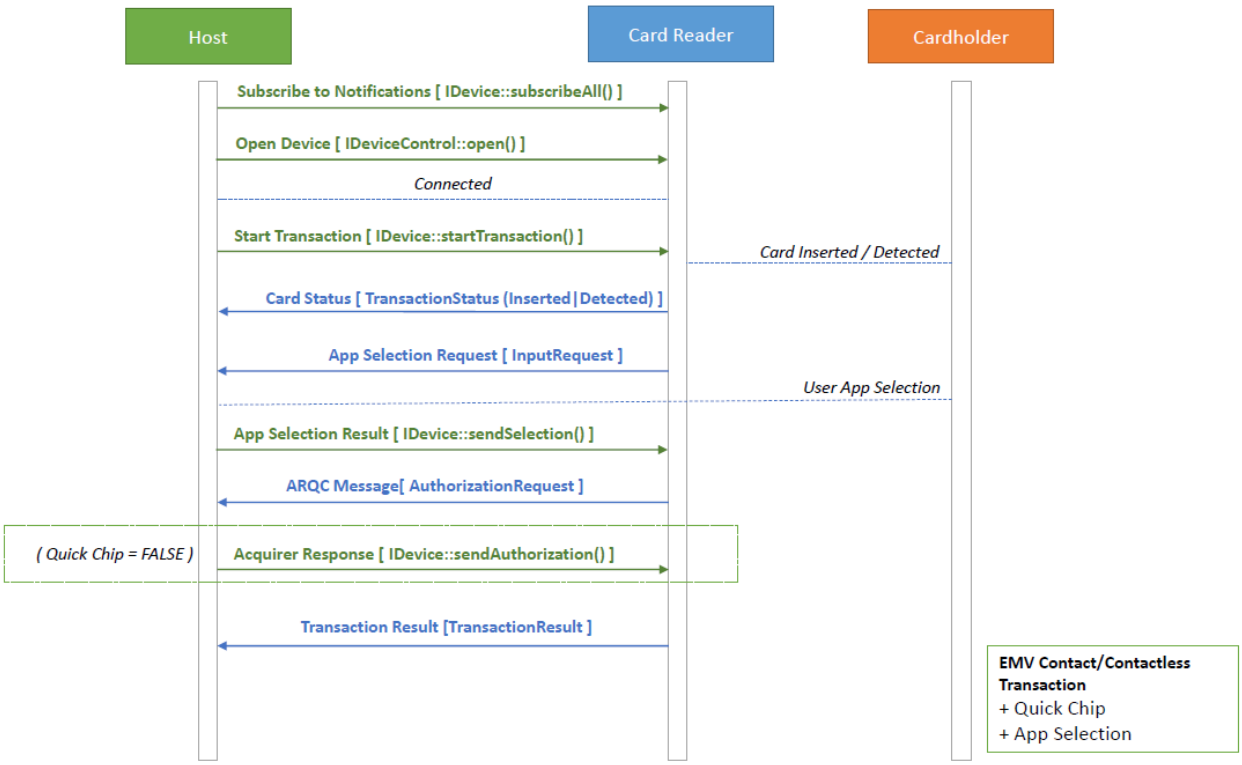
```
      case EVENT_TYPE_DisplayMessage:
        // Display approval message.
        message = data.StringValue;

        // A data size of 0 is an instruction to clear the display.
        if (data.StringValue.length() == 0)
        {
          // Clear the UI display.
        }
    }
}
```

```
virtual void OnEvent(EventType eventType, IData data)
{
  string message;
  switch (eventType);
    {
      case EVENT_TYPE_TransactionResult:
         /* data.ByteArray[0]    – Signature Required
            data.ByteArray [1..2] – Batch Data length
            data.ByteArray [3..n] – remainder contains the Batch Data
            TLV object */

        // Parse the TLV from data[].
        // Abstract Approval status from TLV tag “DFDF1A”.
        // Abstract Signature Required status from TLV tag data[0].
    }
}
```

B.3 Transaction Sequence: Contact/Contactless



B.4 Transaction Sequence: Contact/Contactless + MPPG