MAGTEK®
SECURITY FROM THE INSIDE

MagTek Universal SDK

For MMS Devices
Programmer's Manual (iOS)

November 2025

Manual Part Number:
D998200386-105

REGISTERED TO ISO 9001:2015

Copyright © 2006 - 2025 MagTek, Inc.
Printed in the United States of America

Information in this publication is subject to change without notice and may contain technical inaccuracies or graphical discrepancies. Changes or improvements made to this product will be updated in the next publication release. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of MagTek, Inc.

MagTek® is a registered trademark of MagTek, Inc.
MagnePrint® is a registered trademark of MagTek, Inc.
MagneSafe® is a registered trademark of MagTek, Inc.
Magensa™ is a trademark of MagTek, Inc.
aDynamo™, iDynamo™, and uDynamo™ are trademarks of MagTek, Inc.
eDynamo™, Dynamag, and DynaMAX are trademarks of MagTek, Inc.
mDynamo™, DynaWave™, and tDynamo™ are trademarks of MagTek, Inc.
DynaPro Go™, DynaPro™, and DynaPro Mini™ are trademarks of MagTek, Inc.
DynaFlex™ and DynaProx™ are trademarks of MagTek, Inc.

The Bluetooth® word mark and logos are registered trademarks owned by Bluetooth SIG, Inc. and any use of such marks by MagTek is under license.
Microsoft® and Windows® are registered trademarks of Microsoft Corporation.
iPhone®, iPod touch®, Mac®, and Xcode® are registered trademarks of Apple Inc., registered in the U.S. and other countries. App Store[SM] is a service mark of Apple Inc., registered in the U.S. and other countries. iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used by Apple Inc. under license.
iPad™ is a trademark of Apple Inc.
MAC and macOS are trademarks of Apple Inc., registered in the U.S. and other countries.

EMV® is a registered trademark in the U.S. and other countries and an unregistered trademark elsewhere. The EMV trademark is owned by EMVCo, LLC. The Contactless Indicator mark, consisting of four graduating arcs, is a trademark owned by and used with permission of EMVCo, LLC.

All other system names and product names are the property of their respective owners.

Table 0.1 - Revisions

Rev Number	Date	Notes
100	September 27, 2023	Initial release
101	March 28, 2024	Added support for DynaFlex II Go connection at section 1.5. Added External Accessory support at sections 3.8, to 3.12, 6.14, 6.20, 6.27, and 6.28. Added completion handlers at sections 4.2, 4.15, 6.3, 6.8, 6.10, 6.17, 6.23, 6.26, and 9.10. Added ConfigurationInfo as section 10. Added MTUSDKDelegate as section 14. Removed all instances of getDeviceList().
102	July 23, 2024	Added EnhancedInputRequest to EventType (Section 15.15). Added NFC events to UserEvent (Section 15.17). Added NFC functions to IDevice (Section 4.14, 4.15, 4.16).

103	August 14, 2024	Updated the signature for delegate onDeviceList(). Updated its parameter connectionType to withConnectionType (Section 14.1).
104	May 19, 2025	Added support for MQTT API. Added MQTT methods to CoreAPI at section 3. Added didSystemUpdateState delegate at section 14.2. Added MQTT to ConnectionType at section 15.4. Added SystemState enum at section 15.15.
105	November 12, 2025	Added DisplayAmountForQuickChip to iTransaction at section 11.5.

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

Table of Contents 6
1 Introduction 11
1.1 About the MagTek Sample Code 11
1.2 Nomenclature 11
1.3 System Requirements 11
1.4 Connect via WLAN 12
1.5 Connect via Bluetooth LE 16
2 How to Setup the SDK 17
3 CoreAPI 20
3.1 createDevice 20
3.2 getAPIVersion 22
3.3 getConnectionTypes 22
3.4 getConnectionTypeFromString 22
3.5 isDevice 23
3.6 loadClientCertificate 23
3.7 setDeviceType 23
3.8 setupEADeviceProtocolString 23
3.9 setMQTTBrokerInfo 25
3.10 setMQTTClientID 25
3.11 setMQTTClientCertificateInfo 26
3.12 setMQTTDeviceDiscoveryTimeout 26
3.13 setMQTTPublishTopic 26
3.14 setMQTTQos 26
3.15 setMQTTSubscribeTopic 27
3.16 showConnectedEAAccessoryIfAny 27
3.17 startDiscover 27
3.18 startScanningForPeripherals 27
3.19 stopDiscover 28
3.20 stopScanningForPeripherals 28
3.21 turnEAAccessoryConnectionNotificationsOn 28
3.22 turnEAAccessoryConnectionNotificationsOff 28
4 IDevice (Payment Functions) 29
4.1 cancelTransaction 29
4.2 cancelTransactionWithCompletionHandler 29
4.3 getConnectionInfo 29
4.4 getConnectionState 29
4.5 getDeviceCapability 29
4.6 getDeviceConfiguration 30
4.7 getDeviceControl 30

4.8 getDeviceInfo 30
4.9 deviceName 30
4.10 requestPAN 30
4.11 requestPIN 32
4.12 requestSignatureWithCompletionHandler 33
4.13 sendAuthorization 34
4.14 sendClassicNFCCommand 34
4.15 sendDESFireNFCCommand 37
4.16 sendNFCCommand 38
4.17 sendSelection 42
4.18 startTransaction 43
4.19 subscribeAll 44
4.20 unsubscribeAll 44
5 DeviceCapability 45
5.1 AutoSignatureCapture 45
5.2 BatteryBackedClock 45
5.3 Display 45
5.4 MSRPowerSaver 45
5.5 PaymentMethods 45
5.6 PINPad 45
5.7 Signature 46
5.8 SRED 46
6 IDeviceControl 47
6.1 close 47
6.2 deviceReset 47
6.3 displayMessage 47
6.4 endSession 49
6.5 getInput 49
6.6 open 49
6.7 playSound 49
6.8 resetDeviceWithCompletionHandler 50
6.9 send 50
6.10 sendData 50
6.11 sendExtendedCommand 50
6.12 sendSync 51
6.13 setDateTime 51
6.14 setupEADeviceProtocolString 51
6.15 setLatch 52
6.16 showImage 52
6.17 showImage 52
6.18 showImage 53

6.19 showBarCode 53
6.20 showConnectedEAAccessoryIfAny 54
6.21 startBarCodeReader 55
6.22 startScan 55
6.23 startScanBarcodeWithTimeout 55
6.24 stopBarCodeReader 56
6.25 stopScan 56
6.26 stopScanBarcodeWithCompletionHandler 56
6.27 turnEAAccessoryConnectionNotificationsOn 56
6.28 turnEAAccessoryConnectionNotificationsOff 56
7 ConnectionInfo 58
7.1 getAddress 58
7.2 getConnectionType 58
7.3 getDeviceType 58
7.4 getCertificateInfo 58
7.5 initWithDeviceType 59
8 DeviceInformation 62
8.1 deviceModel 62
8.2 deviceName 62
8.3 deviceSerial 62
8.4 initWithName 62
9 IDeviceConfiguration 63
9.1 getChallengeToken 63
9.2 getConfigInfo 63
9.3 getDeviceInfo 63
9.4 getFile 64
9.5 getKeyInfo 64
9.6 sendFile 64
9.7 sendImage 65
9.8 sendSecureFile 65
9.9 setConfigInfo 66
9.10 setDisplayImage 66
9.11 updateFirmware 67
9.12 updateKeyInfo 67
10 ConfigurationInfo 68
10.1 configType 68
10.2 OidAndValue 68
10.3 fromASN1Oid 68
11 Data Classes 69
11.1 BarCodeData 69
11.2 CertificateInfo 69

11.3 IData ... 69
11.4 IResult ... 70
11.5 ITransaction ... 70
12 IEventSubscriber Delegate ... 75
12.1 OnEvent ... 75
13 IConfigurationCallback Delegates ... 76
13.1 OnCalculateMAC ... 76
13.2 OnProgress ... 76
13.3 OnResult ... 76
14 MTUSDKDelegate Delegate ... 78
14.1 onDeviceList ... 78
14.2 didSystemUpdateState ... 78
15 Enumerations ... 79
15.1 BarCodeFormat ... 79
15.2 BarCodeType ... 79
15.3 ConnectionState ... 79
15.4 ConnectionType ... 79
15.5 DataEntryType ... 81
15.6 DeviceEvent ... 81
15.7 DeviceFeature ... 81
15.8 DeviceType ... 81
15.9 EventType ... 82
15.10 FeatureStatus ... 83
15.11 ImageType ... 84
15.12 InfoType ... 84
15.13 PaymentMethod ... 85
15.14 StatusCode ... 85
15.15 SystemState ... 85
15.16 TransactionStatus ... 86
15.17 OperationStatus ... 87
15.18 UserEvent ... 88
15.19 VASMode ... 88
15.20 VASProtocol ... 90
Appendix A API Walk Through ... 91
A.1 CoreAPI Walk Trough ... 91
A.2 IDevice Walk Through ... 93
A.2.1 Handling Events ... 93
A.3 IDeviceControl Walk Through ... 97
A.4 ConnectionInfo Walk Through ... 98
A.5 IDeviceCapability Walk Through ... 99
A.6 IDeviceConfiguration Walk Through ... 100

A.6.1 Handling Events....101
Appendix B EMV Transaction Flow 102
B.1 Flow Chart - QuickChip....102
B.2 Sample Code - QuickChip....103
B.3 Flow Chart – Signature Capture....105
B.4 Sample Code – Signature Capture....106
B.5 Flow Chart – With ARPC....109
B.6 Sample Code – With ARPC....110
B.7 MSR Fallback Flow....114

1 Introduction

This document provides instructions for software developers who want to create iOS software solutions that include MagTek devices connected to an iOS host. MagTek Universal SDK (MTUSDK) incorporates MagTek SCRA and MagTek PIN Pad SCRA devices into one SDK. This document is part of a larger library of documents designed to assist MagTek device implementers.

The following are absolute prerequisites:

- ***D998200383 DynaFlex Family Programmer's Manual (COMMANDS)***
- ***D998200382 DynaFlex and DynaFlex Pro Installation and Operation Manual***

1.1 About the MagTek Sample Code

The sample code provides Swift demonstration source code and a reusable MTUSDK library that provides developers of custom software solutions with an easy-to-use interface for MagTek devices. Developers can distribute the MTUSDK library to customers or distribute internally as part of an enterprise solution.

1.2 Nomenclature

- **Device** refers to the MagTek devices that receives and responds to command set.
- **Host** refers to the piece of general-purpose electronic equipment the device is connected or paired to, which sends data to and receives data from the device. Host types include but not limited to PC and Mac computers, tablets, and smartphones. When "host" must be used differently, it is qualified as something specific, such as "USB host."
- **User** in this document generally refers to the **cardholder**.

1.3 System Requirements

Tested operating systems:

- iOS 13 and above.
- Xcode 14 and above.

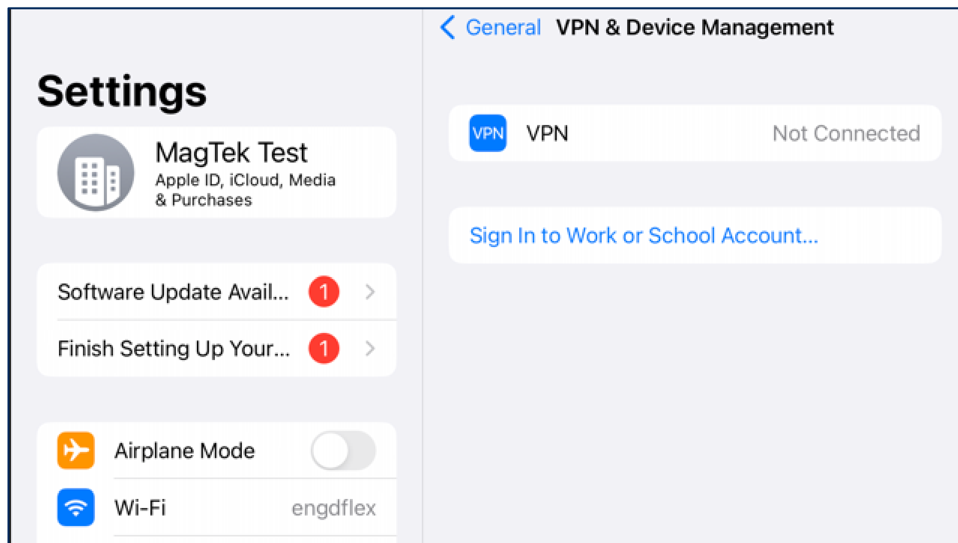
1.4 Connect via WLAN

When connecting to the DynaFlex II PED device by WebSocket, a client certificate and its certificate chain must be installed on the iOS device.

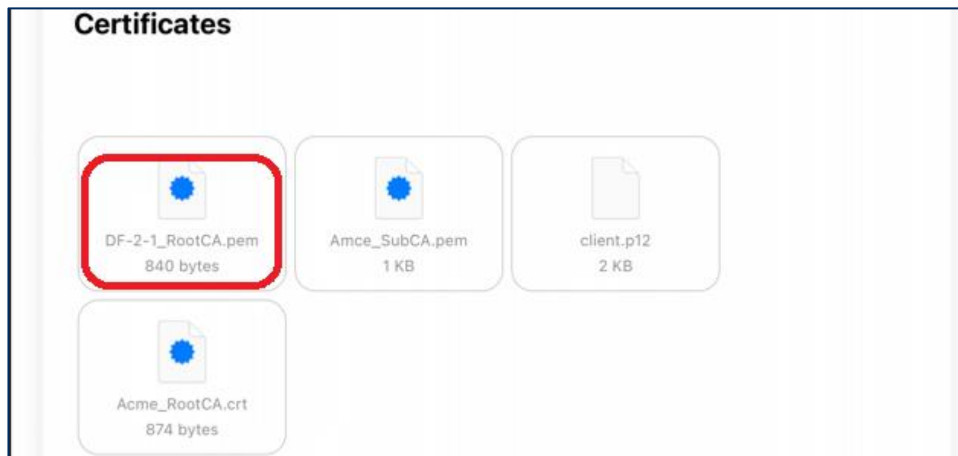
The Root Certificate, Sub CA Certificate, and Client private key Certificate as referenced in document ***D998200550 DynaFlex TLS Certificate Installation Manual***. The client private key certificate (e.g. filename client.p12) must be accessible to the custom software that makes a secure WebSocket connection to the DynaFlex II PED.

After creating the certificates and transferring them in the iOS device, load the certificate chain into the iOS device by follow these steps. The certificate names shown here are for reference only.

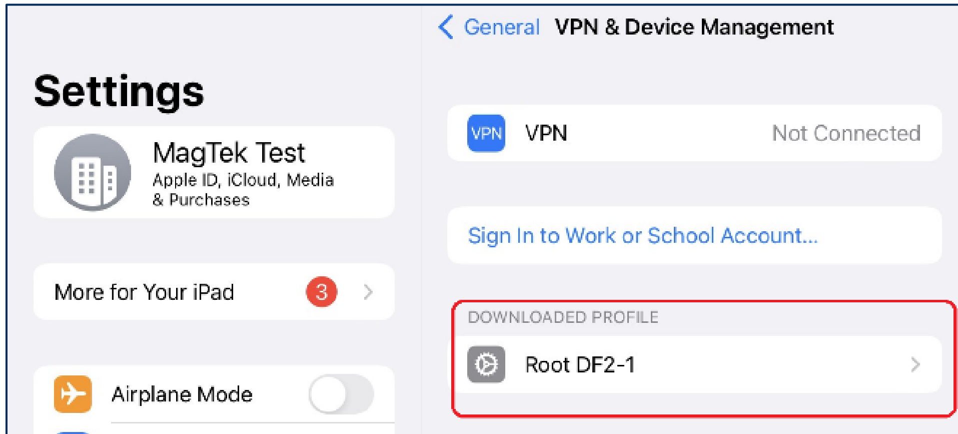
1) Delete the previously installed certificates prior to beginning this procedure. If the certificate chain is current, skip this installation section. View the certificates by going to Settings → General → VPN & Device Management.



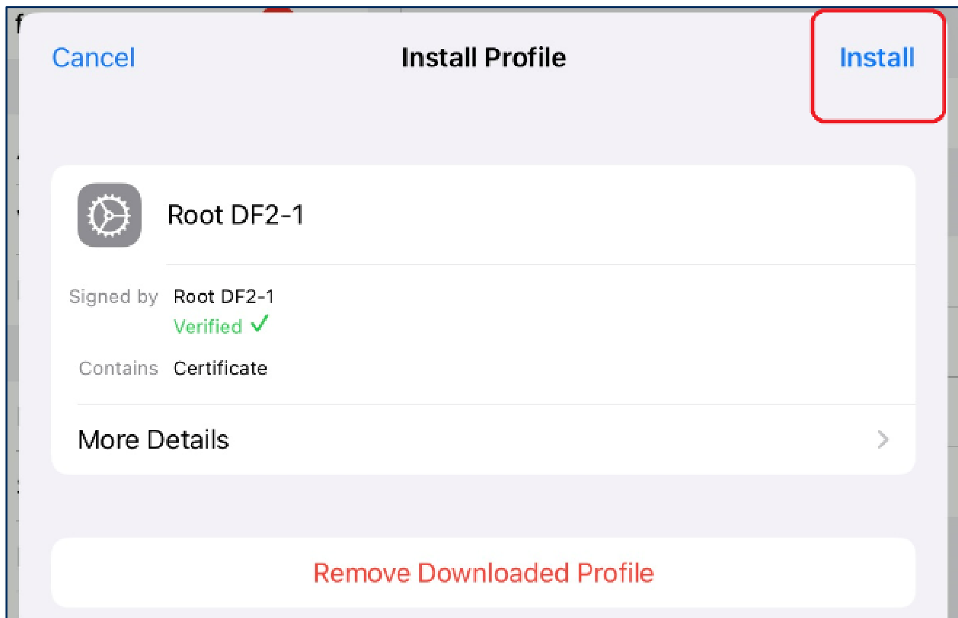
2) Copy the certificate chain to the iOS device. Touch the Root certificate DF-2-1_RootCA.pem to download to iOS device.



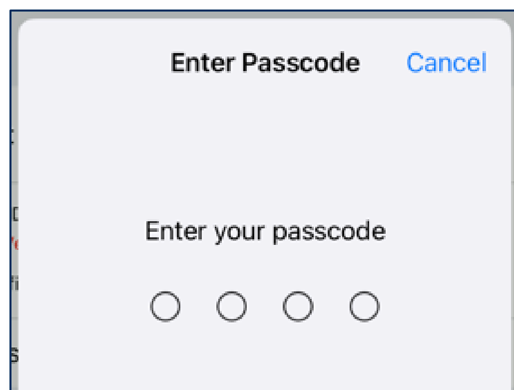
3) Go to Settings > General >VPN & Device Management > Select Root DF2-1



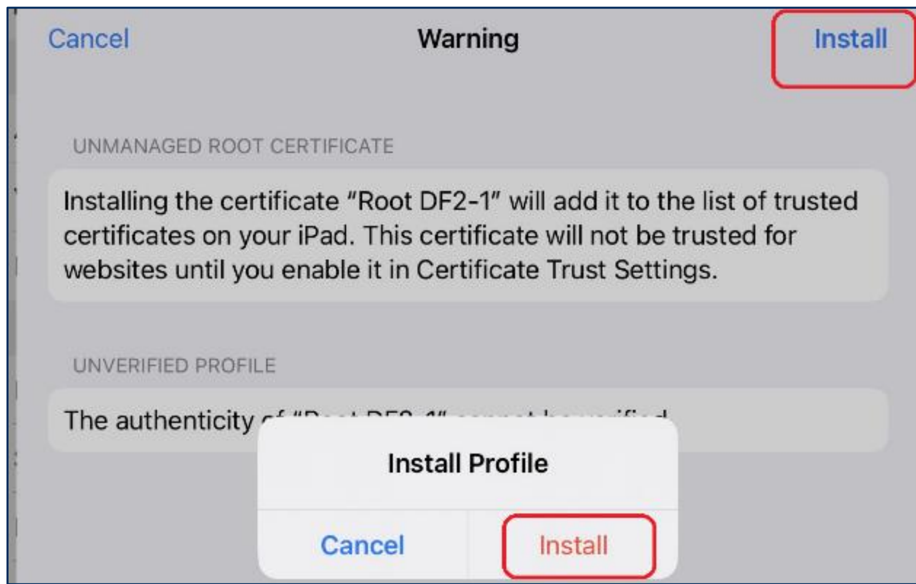
4) Select Install.



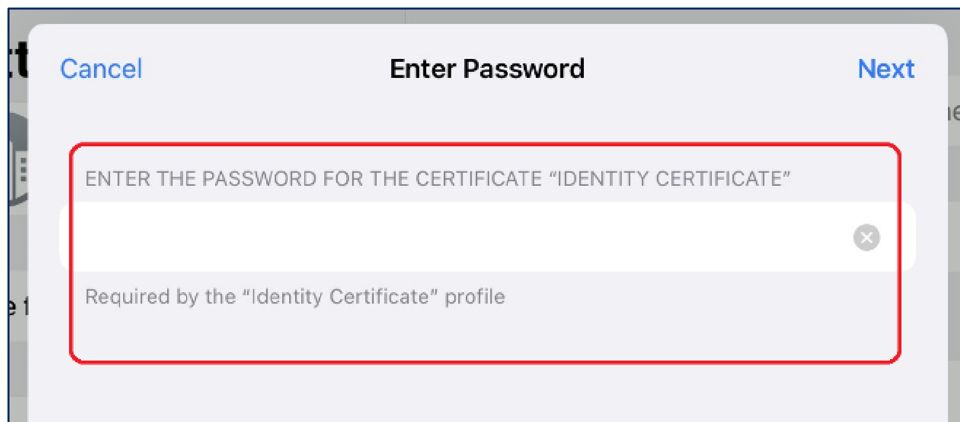
5) Enter Passcode.



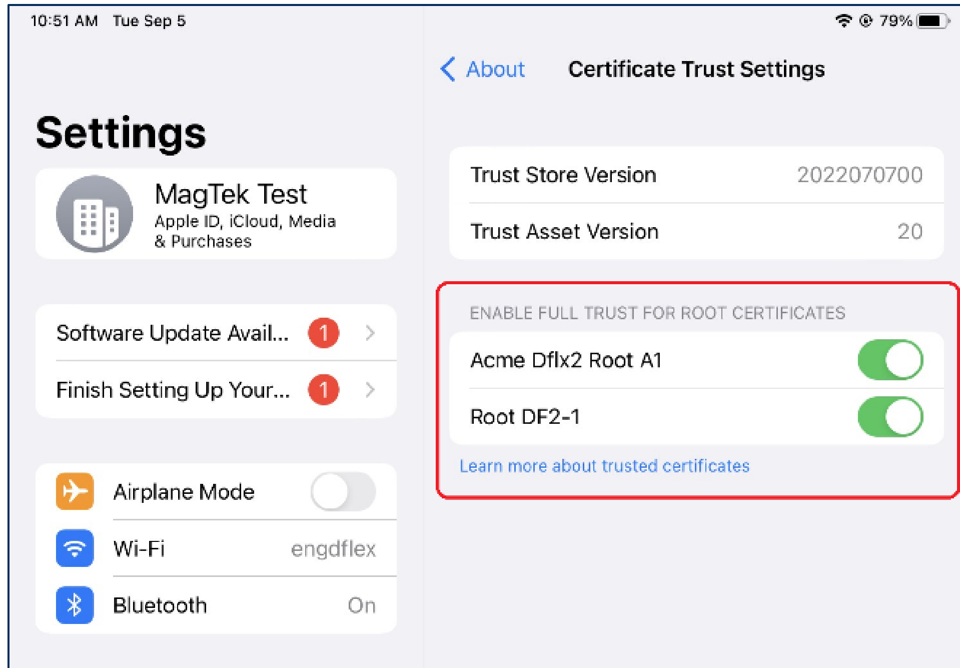
6) Press Install twice.



7) Repeat the same steps above for the SubCA certificate Amce_SubCA.pem and for the Client certificate Client.p12. For Client, iOS will ask for an additional certificate password, enter the password used during the creation of the Client certificate when prompted, and then press Next.



8) Enable Certificates by going to Settings → About → Certificate Trust Settings



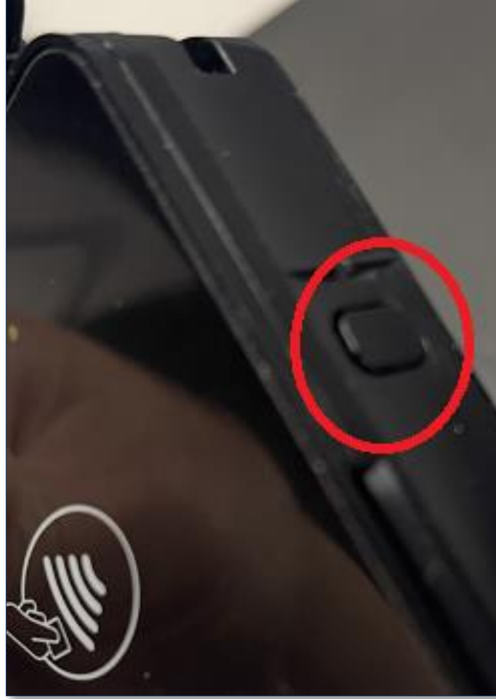
9) When building the custom app, the Client certificate must be loaded into the data member of CertificateInfo during the call to createDevice(). See below as an example to access the client certificate.

```
252    func createWebSocketDevice() -> IDevice? {
253        guard let currentSelectedDeviceAddress = currentSelectedDeviceAddress else { return nil }
254
255        let clientCert = Bundle.main.url(forResource: "client", withExtension: "p12")!
256        let certData = try! Data(contentsOf: clientCert)
257
258        if currentSelectedDeviceAddress.protocolCheck("wss") {
259            return CoreAPI.createDevice(
260                MTU_DeviceType_MMS,
261                connection: MTU_ConnectionType_WEBSOCKET,
262                address: currentSelectedDeviceAddress,
263                model:"DynaFlex",
264                name: currentSelectedDeviceAddress,
265                serial:"B123456",
266                cert: CertificateInfo(format: "PKCS12", data: certData, password: "")
267            )
268        }
269        else if currentSelectedDeviceAddress.protocolCheck("ws") {
270            return CoreAPI.createDevice(
271                MTU_DeviceType_MMS,
272                connection: MTU_ConnectionType_WEBSOCKET,
273                address: currentSelectedDeviceAddress,
274                model:"DynaFlex",
275                name: currentSelectedDeviceAddress,
276                serial:"B123456"
277            )
278        }
279
280        return nil
281    }
```

1.5 Connect via Bluetooth LE

The following instructions are for Bluetooth pairing the DynaFlex II Go.

1) Place the device into Pairing Mode by pressing and holding the power button on device until 4 beeps and release the button. The 4th LED will blink green.



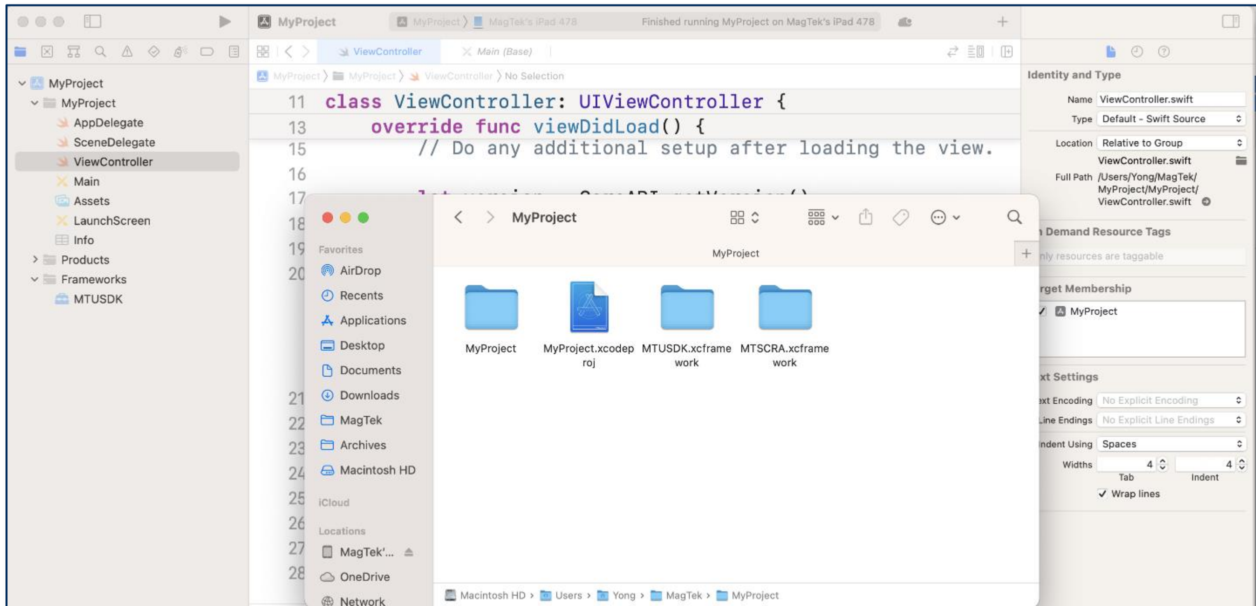
2) When in pairing mode and the first time connecting to device, the host will prompt for a pairing request. Enter the pairing code. The default code is "000000".

2 How to Setup the SDK

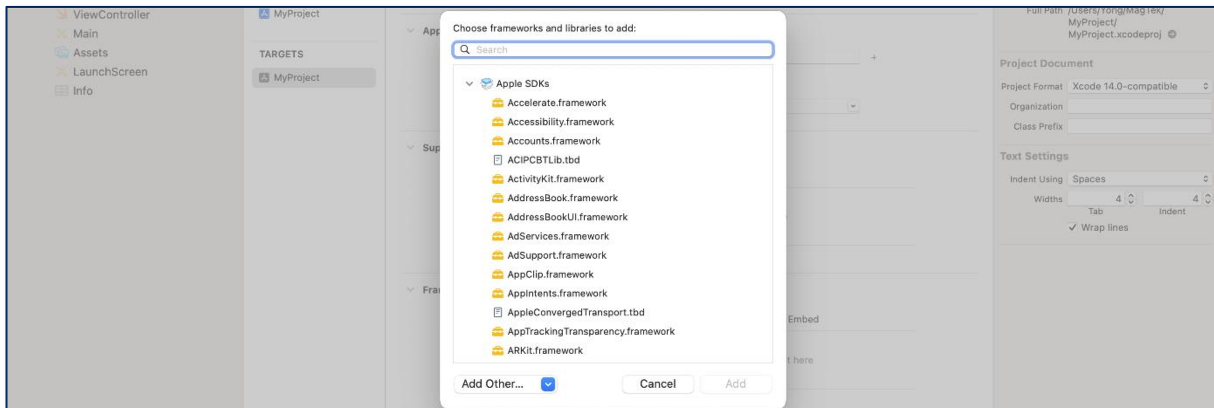
To set up the MagTek Universal SDK library, download and install the MTUSDK available from MagTek.com.

To build out demo code, follow these steps:

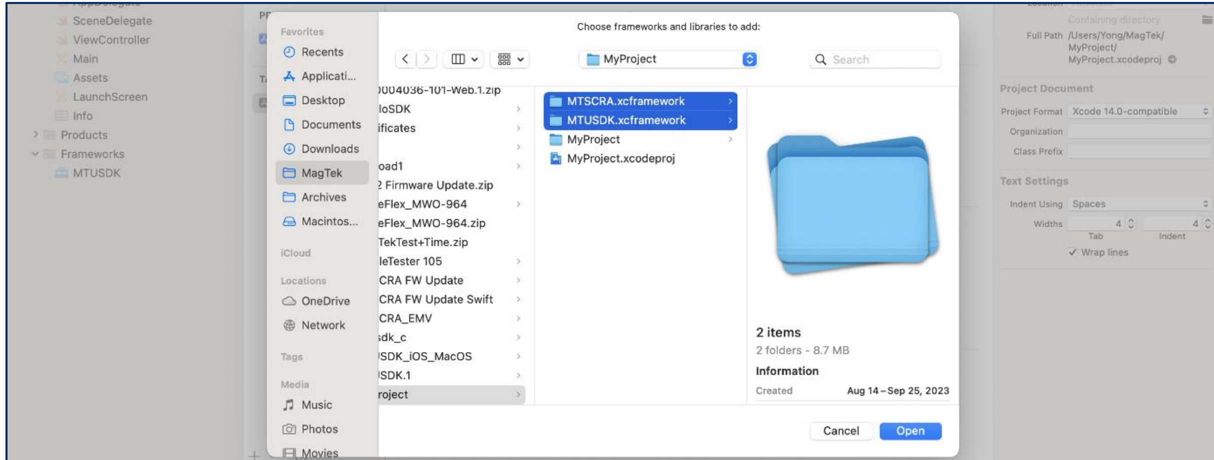
1) Locate and copy the framework files to your project folder. iOS files is under folder ios-arm64 of the SDK package.



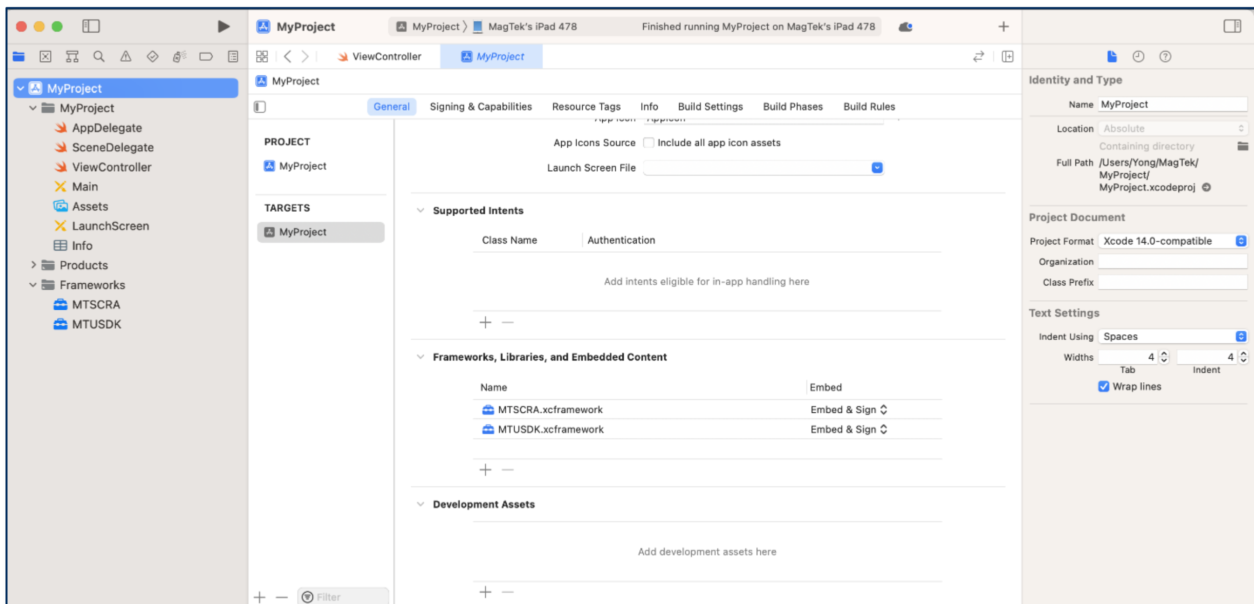
2) Click your project and select “General”. In the section “Frameworks, Libraries, and Embedded Content”, click “+”.



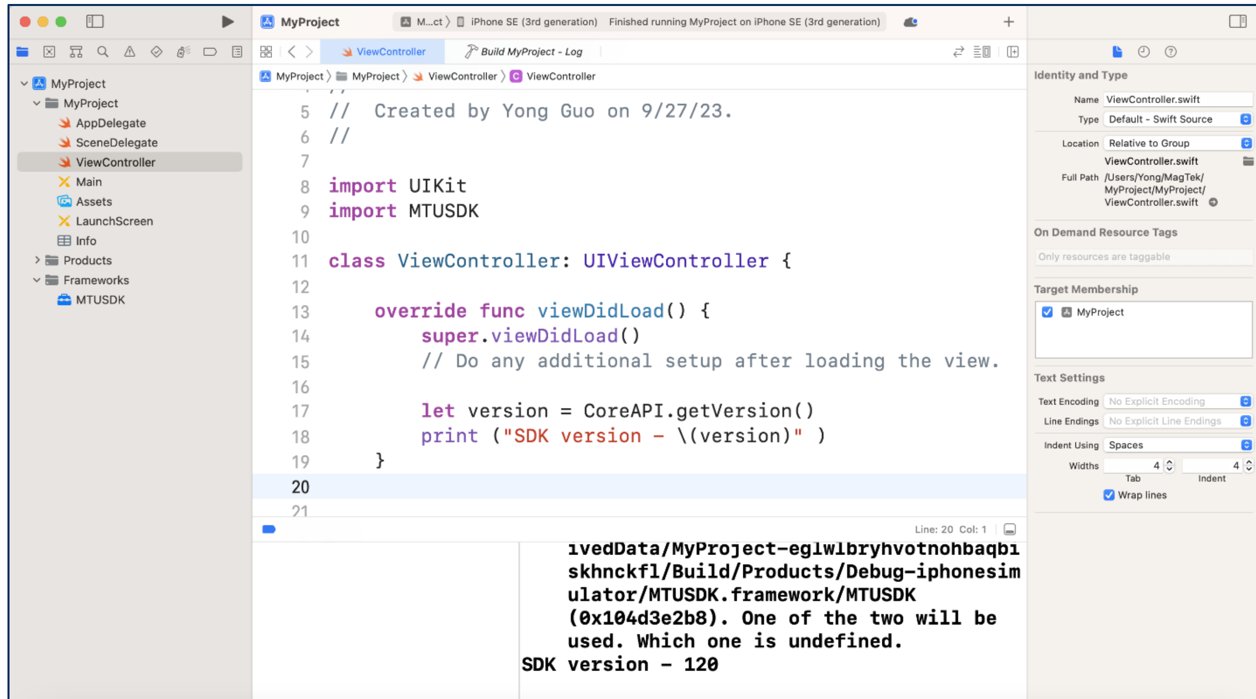
3) Pick the framework files - “MTUSDK.xcframework” and “MTSCRA.xcframework” (MTSCRA is referenced in MTUSDK).



4) Validate that the framework is embedded in your project.



5) Add code and build.


MyProject
iPhone SE (3rd generation)
Finished running MyProject on iPhone SE (3rd generation)
ViewController
Build MyProject - Log
MyProject
MyProject
AppDelegate
SceneDelegate
ViewController
Main
Assets
LaunchScreen
Info
Products
Frameworks
MTUSDK
5 // Created by Yong Guo on 9/27/23.
6 //
7
8 import UIKit
9 import MTUSDK
10
11 class ViewController: UIViewController {
12
13   override func viewDidLoad() {
14   super.viewDidLoad()
15   // Do any additional setup after loading the view.
16
17   let version = CoreAPI.getVersion()
18   print ("SDK version - \(version)")
19   }
20
Line: 20 Col: 1
ivedData/MyProject-eglwlbryhvotnohbaqbi
skhnckfl/Build/Products/Debug-iphonesim
ulator/MTUSDK.framework/MTUSDK
(0x104d3e2b8). One of the two will be
used. Which one is undefined.
SDK version - 120
Identity and Type
Name ViewController.swift
Type Default - Swift Source
Location Relative to Group
ViewController.swift
Full Path /Users/Yong/MagTek/MyProject/MyProject/ViewController.swift
On Demand Resource Tags
Only resources are taggable
Target Membership
MyProject
Text Settings
Text Encoding No Explicit Encoding
Line Endings No Explicit Line Endings
Indent Using Spaces
Widths 4 4
Tab Indent
Wrap lines

3 CoreAPI

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

3.1 createDevice

This function creates an instance of IDevice. Once created, the API's of MagTek Universal SDK may be utilized.

```
(IDevice*) createDevice: (MTU_DeviceType) deviceType
             connection: (MTU_ConnectionType) connectionType
                address: (NSString*) address
                  model: (NSString*) model
                   name: (NSString*) name
                 serial: (NSString*) serialNumber;

(IDevice*) createDevice: (MTU_DeviceType) deviceType
             connection: (MTU_ConnectionType) connectionType
                address: (NSString*) address
                  model: (NSString*) model
                   name: (NSString*) name
                 serial: (NSString*) serialNumber
                   cert: (CertificateInfo*) cert;
```

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

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>address</td><td>Address for the device.

For Ethernet devices, address should be in the form:
<code>IP://IP-Address:PORT</code>
for example, <code>IP://10.57.10.180:26</code>

For Wireless devices, address should be in the form:
<code>TLS12://TLSDEVICESERIALNUMBER</code>
<code>TLS12TRUST://TLSDEVICESERIALNUMBER</code>
for example, <code>TLS12://TLS99261829170E0810</code>
<code>TLS12TRUST://TLS99261829170E0810</code>

For Bluetooth LE devices, address should be in the form:
<code>BLEEMV://DEVICENAME</code>
for example, <code>BLEEMV://DynaPro Go-EB66</code>

For WebSocket or Secure WebSocket devices, address should be in the form:
<code>ws://IP-Address or Hostname</code>
<code>wss://IP-Address or Hostname</code>
for example, <code>ws://192.168.1.150</code>
<code>wss://b512345.magtek.com</code>

The client private key certificate (e.g. filename client.p12) must be accessible to the custom software that makes a secure WebSocket (wss://) connection to the DynaFlex II PED.

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]</td></tr>
<tr><td>model</td><td>Model name for the device.</td></tr>
<tr><td>name</td><td>Unique name for the device to distinguish between multiple devices of the same model.</td></tr>
<tr><td>serialNumber</td><td>Serial number for the device.</td></tr>
</table>

Parameter	Description
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*** The client private key certificate (client.p12) must be accessible to the custom software that makes a secure WebSocket (wss://) connection to the DynaFlex II PED. The client.p12 certificate is not needed if the connection type is not secure WebSocket (ws://)

Return Value:
Returns an IDevice*.

3.2 getAPIVersion

This function returns the API version.

```
(NSInteger) CoreAPI.getAPIVersion;
```

Return Value:
Returns an integer representing the API version.

3.3 getConnectionTypes

This function returns connection types from a device type.

```
(NSArray<NSString*>*)getConnectionTypes: (MTU_DeviceType) deviceType;
```

Parameter	Description
deviceType	An enum for the type of MagTek readers which the SDK will control.

Return Value:
Returns an array of NSString.

3.4 getConnectionTypeFromString

This function returns a connection type from a string of connection type.

```
(MTU_ConnectionType)getConnectionTypeFromString: (NSString*)
connectionTypeString;
```

Parameter	Description
connectionTypeString	A string for the type of connection to the MagTek readers which the SDK will control.

Return Value:
Returns an MTU_ConnectionType.

3.5 isDevice

This function checks if this is a valid device.

```
(Boolean) isDevice: (MTU_DeviceType) deviceType
        enumerable: (MTU_ConnectionType) connectionType;
```

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

Return Value:
Returns true if device is valid.

3.6 loadClientCertificate

This function loads a certificate for mTLS. It can used for Web Socket and MQTT.

```
(int)loadClientCertificate:(NSString* _Nonnull)format
                      data:(NSData* _Nonnull)data
                  password:(NSString* _Nonnull)password;
```

Parameter	Description
format	Certificate format. Use "PKCS12"
data	Contents of PKCS12 file
password	Password for PKCS12 file

Return Value:
None

3.7 setDeviceType

This function sets the device type. This must be called before scanning peripherals for BLE or setting up EA accessories for iAP2.

```
(void) setDeviceType: (MTU_DeviceType) deviceType
   andConnectionType: (MTU_ConnectionType) connectionType;
```

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

Return Value:
None

3.8 setupEADeviceProtocolString

This function sets the external accessory protocol string.

```
(void) setupEADeviceProtocolString: (NSString *)protocolString;
```

Parameter	Description
protocolString	Name of the external accessory protocol string.

Return Value:
None

3.9 setMQTTBrokerInfo

This function sets the MQTT (Message Queuing Telemetry Transport) broker information. Call prior to device discovery.

```
(void) setMQTTBrokerInfo: (nonnull NSString*) url
                username: (nullable NSString*) username
                password: (nullable NSString*) password;
```

Parameter	Description
url	URL included the port. Support URLs: • TCP: “test.mosquitto.org”, “test.mosquitto.org:1883”, “mqtt://test.mosquitto.org:1883”, “mqtt://broker.emqx.io:1883” • TCP (Authenticated): “test.mosquitto.org:1884”, “mqtt://test.mosquitto.org:1884” • TCP (Encrypted): “mqtts://test.mosquitto.org:8886”, “mqtts://broker.emqx.io:8883” • TCP (Encrypted & Authenticated): “mqtts://test.mosquitto.org:8885” • WebSocket: “ws://test.mosquitto.org:8080”, “ws://broker.emqx.io:8083” • WebSocket (Encrypted): “wss://test.mosquitto.org:8081” , “wss://broker.emqx.io:8084” • WebSocket (Authenticated): “ws://test.mosquitto.org:8090” • WebSocket, (Encrypted & Authenticated): “wss://test.mosquitto.org:8091”
username	Username
password	Password

Return Value:
None

3.10 setMQTTClientID

This function sets the MQTT client ID to establish a connection.

```
(void) setMQTTClientID: (nonnull NSString*) clientID;;
```

Parameter	Description
clientID	Client ID. If not set, the default value is [HostName]-[RandomUUID]

Return Value:
None

3.11 setMQTTClientCertificateInfo
This function sets the MQTT certificate information.

```
(void)setMQTTClientCertificateInfo:(CertificateInfo*) certificateInfo;
```

Parameter	Description
certificateInfo	If client certificate is required when initiating a connection to the MQTT broker, this shall be used to establish the connection. If not set, the default value is NULL.

Return Value:
None

3.12 setMQTTDeviceDiscoveryTimeout
This function sets the MQTTDeviceDiscoveryTimeout value. RFU.

```
(void) setMQTTDeviceDiscoveryTimeout : (int) Seconds;
```

Parameter	Description
Seconds	Time out in milliseconds. Not implemented. Reserved for future use.

Return Value:
None

3.13 setMQTTPublishTopic
This function sets the base value for MQTTPublishTopic.. When connected to the MQTT broker, the composed topic is in the format:
"<basetopic>/<DeviceID>/MMSMessage".

```
(void) setMQTTPublishTopic: (nonnull NSString*) topic;
```

Parameter	Description
topic	Base topic for which to publish messages. Full topic is composed by the SDK.

Return Value:
None

3.14 setMQTTQos
This function sets the MQTTQoS value.

```
(void) setMQTTQoS : (int) qos;
```

Parameter	Description
qos	The quality of service level for publishing messages. Range: 0 = At most once (default) 1 = At least once 2 = Exactly once

Return Value:
None

3.15 setMQTTSubscribeTopic

This function sets the base value for MQTTSubscribeTopic. When connected to the device, the composed topic is in the format:
"<basetopic>/<DeviceID>/MMSMessage".

Example:
Base topic = "MagTek/Server/DynaFlexIIPED/"
Full topic = "MagTek/Server/DynaFlexIIPED/B51E72D/MMSMessage"

```
(void) setMQTTSubscribeTopic : (nonnull NSString*) topic;
```

Parameter	Description
topic	Base topic for which to subscribe for messages. Full topic is composed by the SDK.

Return Value:
None

3.16 showConnectedEAAccessoryIfAny

This function is to show any connected EA accessories.

```
(void) showConnectedEAAccessoryIfAny;
```

Return Value:
None

3.17 startDiscover

This function begins the discovery for devices and peripherals based in deviceType and connectionType. Found devices and peripherals are returned at the **onDeviceList**() delegate.

```
(void) startDiscover;
```

Return Value:
None

3.18 startScanningForPeripherals

This function begins the scanning for peripherals. Found periherals are returned at the **onDeviceList**() delegate.

```
(void) startScanningForPeripherals;
```

Return Value:
None

3.19 stopDiscover

This function ends the discovery for devices and peripherals based in deviceType and connectionType.

```
(void) stopDiscover;
```

Return Value:
None

3.20 stopScanningForPeripherals

This function ends the scanning for peripherals.

```
(void) stopScanningForPeripherals;
```

Return Value:
None

3.21 turnEAAccessoryConnectionNotificationsOn

This function turns on the connection notifications for EA accessories.

```
(void) turnEAAccessoryConnectionNotificationsOn;
```

Return Value:
None

3.22 turnEAAccessoryConnectionNotificationsOff

This function turns off the connection notifications for EA accessories.

```
(void) turnEAAccessoryConnectionNotificationsOff;
```

Return Value:
None

4 IDevice (Payment Functions)

Create an instance of the IDevice from CoreAPI.setDeviceType() and the onDeviceList() event, or from CoreAPI.createDevice(). Then use the functions described in this chapter.

4.1 cancelTransaction

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

```
(BOOL) cancelTransaction;
```

Return Value:
Returns true if canceled. Otherwise, returns false.

4.2 cancelTransactionWithCompletionHandler

This function cancels a transaction with a completion handler. A transaction can only be canceled before a payment method is presented.

```
(void) cancelTransactionWithCompletionHandler: (BooleanCallback)
handler;
```

Parameter for data	Description
handler	Completion handler. If the operation is successful, the completion handler will return true, otherwise false. typedef void(^BooleanCallback)(Boolean resultFlag);

Return Value:
None.

4.3 getConnectionInfo

This function retrieves the connection information of the device.

```
(nullable ConnectionInfo*) getConnectionInfo;
```

Return Value:
Returns `ConnectionInfo*`

4.4 getConnectionState

This function retrieves the connection state of the device.

```
(MTU_ConnectionState) getConnectionState;
```

Return Value:
Returns `MTU_ConnectionState`

4.5 getDeviceCapability

This function retrieves the capability of the device.

```
(nullable DeviceCapability*) getDeviceCapability;
```

Return Value:
Returns `DeviceCapability*`

4.6 getDeviceConfiguration

This function allows the host to get an **IDeviceConfiguration** to configure the device.

```
(nullable IDeviceConfiguration*) getDeviceConfiguration;
```

Return Value:
Returns `IDeviceConfiguration*`.

4.7 getDeviceControl

This function retrieves the device control interface to the device.

```
(nullable IDeviceControl*) getDeviceControl;
```

Return Value:
Returns `IDeviceControl*`

4.8 getDeviceInfo

This function returns an information class of the device.

```
(nullable DeviceInformation*) getDeviceInfo;
```

Return Value:
Returns `DeviceInformation*`

4.9 deviceName

This property returns the name of the device assigned from createDevice().

```
@property (nonatomic, readonly, getter=getDeviceName) NSString*
deviceName;
```

Return Value:
Returns the name of the device.

4.10 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.

```
(void) requestPAN:(PANRequest*) panRequest
```

```
          withPIN: (nullable PINRequest*) pinRequest
completionHandler: (BooleanCallback) handler;
```

Parameter	Description
panRequest	PANRequest. See table below.
pinRequest	PINRequest. See table below.
handler	Completion handler. If the operation is successful, the completion handler will return true, otherwise false. typedef void(^BooleanCallback)(Boolean resultFlag);

PANRequest:

Parameter	Type	Description
Timeout	Byte	Wait time in seconds.
PaymentMethods	MTU_ PaymentMethod	List of the PaymentMethod enumeration. Usage: `MSR` - For magnetic stripe cards. `Contact` - For EMV chip cards. `Contactless` - For NFC contactless cards. `ManualEntry` - For user to manually enter transaction data without any card access. When set to ManualEntry, the other payment methods do not apply. `BarCode` - Reserved for future use. `AppleVAS` - Reserved for future use. `BarCodeEncryped` - Reserved for future use. `NFC` - For NFC MiFare. `GoogleVAS` - Reserved for future use.

PINRequest:

Parameter	Type	Description
Timeout	Byte	Wait time in seconds.
PINMode	Byte	PIN mode 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	Byte	Minimum length of accepted PIN (>= 4).
MaxLength	Byte	Maximum length of accepted PIN (=< 12).

Parameter	Type	Description
Tone	Byte	Tone to play when prompting for the PIN. Usage: `0x00` - No sound `0x01` - One beep `0x02` - Two beeps
Format	Byte	ISO format for the PIN block. `0x00` - ISO Format 0 `0x01` - ISO Format 1 `0x03` - ISO Format 3 `0x04` - ISO Format 4
PAN	NString*	First 12 digits of the Primary Account Number. Leave blank if not required by the ISO format for the PIN block.

Return Value:
None.

4.11 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.

```
(void) requestPIN: (PINRequest*) pinRequest
completionHandler: (BooleanCallback) handler;
```

Parameter	Description
pinRequest	PINRequest. See table below.
handler	Completion handler. If the operation is successful, the completion handler will return true, otherwise false. typedef void(^BooleanCallback)(Boolean resultFlag);

PINRequest :

Parameter	Type	Description
Timeout	Byte	Wait time in seconds.

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

Return Value:
None.

4.12 requestSignatureWithCompletionHandler

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

```
(void) requestSignatureWithCompletionHandler: (BooleanCallback)
handler;
```

Parameter	Description
handler	Completion handler. If the operation is successful, the completion handler will return true, otherwise false. typedef void(^BooleanCallback)(Boolean resultFlag);

Return Value:
None.

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().

```
(BOOL) sendAuthoriation: (IData*) data;
```

Parameter	Description
data	Contains ARPC blob.

Return Value:
Returns true if successful. Otherwise, returns false.

4.14 sendClassicNFCCommand

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

```
(BOOL) sendClassicNFCCommand:(IData*) data
                 lastCommand: (BOOL) lastCommand
                     encrypt: (BOOL) Encrypt;
```

Parameter	Description
data	Command to send to the NFC tag. 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.

Parameter	Description
Encrypt	Determines if data returned is to be encrypted. `true` = Encrypt data `false` = Do not encrypt data (Default)

Return Value:
Returns true if successful. Otherwise, returns false.

NFC Classic Commands

Command	Length	Field Value
Mifare Read	11	Byte 0 – 0x30 – Read Command Byte 1- Sector Number to Read Byte 2 – Start Block Number Byte 3 – End Block Number Byte 4 – Key Type, 0 = A, 1 = B Byte 5 to 10 = 6 Byte Key
Mifare Write	var	Byte 0 – 0xA0 – Write Command Byte 1- Sector Number to Write Byte 2 – Start Block Number Byte 3 – End Block Number Byte 4 – Key Type 0 = A, 1 = B Byte 5 to 10 = 6 Byte Key Byte 11 to x = Variable length Byte Data (16 bytes per block)
Mifare Increment	14	Byte 0 – 0xC1 – Increment Command Byte 1 – Source Sector Number Byte 2- Source Block number Byte 3 – Key Type 0 = A, 1 = B Byte 4 to 9 = 6 Byte Key Byte 10 to 13 = 4 Byte Operand
Mifare Decrement	14	Byte 0 – 0xC0 – Decrement Command Byte 1 – Source Sector Number Byte 2- Source Block number Byte 3 – Key Type 0 = A, 1 = B Byte 4 to 9 = 6 Byte Key Byte 10 to 13 = 4 Byte Operand
Mifare Restore	10	Byte 0 – 0xC2 – Restore Command Byte 1 – Source Sector Number Byte 2 - Source Block number Byte 3 – Key Type 0 = A, 1 = B Byte 4 to 9 = 6 Byte Key

Command	Length	Field Value
Mifare Transfer	10	Byte 0 – 0xB0 – Write the value from the Transfer Buffer into destination block number Byte 1 – Destination Sector Number Byte 2 - Destination Block number Byte 3 – Key Type 0 = A, 1 = B Byte 4 to 9 = 6 Byte Key

Response Data For NFC Mifare DESFire Light Tag Type 4

<table>
<tr><th>Tag</th><th>Len</th><th>Value / Description</th></tr>
<tr><td>81</td><td>var</td><td>Tag Response Code.
Byte 0 = 0x00 = Success
Byte 0 = 0x01 = I/O Failed
Byte 0 = 0x02 Authentication Failed

Byte 1 = 0x01 = Block that Failed (optional)</td></tr>
<tr><td>82</td><td>var</td><td>Encryption Control Payload tag data.
• Data of R-APDU

If unencrypted:
<table>
<tr><th>Tag</th><th>Len</th><th>Value / Description</th></tr>
<tr><td>FC</td><td>var</td><td>NFC Data Container</td></tr>
<tr><td>/DF7A</td><td>var</td><td>NFC Data</td></tr>
</table>

If encrypted:
<table>
<tr><th>Tag</th><th>Len</th><th>Value / Description</th></tr>
<tr><td>/DFDF59</td><td>var</td><td>Encrypted Data Primitive to be decrypted.</td></tr>
<tr><td>/DFDF50</td><td>var</td><td>Encrypted Data KSN</td></tr>
<tr><td>/DFDF51</td><td>01</td><td>Encrypted Data Encryption Type</td></tr>
</table>
</td></tr>
</table>

Example Unencrypted Payload

```
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

```
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
```

4.15 sendDESFireNFCCommand

This function sends a command to an NFC Mifare DESFire Light Tag Type 4. The NFC tag must first be activated by calling startTransaction() with NFC enabled.

```
(BOOL) sendDESFireNFCCommand:(IData*) data
                lastCommand: (BOOL) lastCommand
                    encrypt: (BOOL) Encrypt;
```

Parameter	Description
data	Command to send to the NFC tag. See DESFire Data Sheet (MF2DLHX0). Should follow ISO 7816-4 APDU format. For details of the command see ***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:
Returns true if successful. Otherwise, returns false.

Response Data For NFC Mifare DESFire Light Tag Type 4

Tag	Len	Value / Description
81	02	Tag Response (SW1 SW2). See DESFire Data Sheet (MF2DLHX0). Should follow ISO 7816-4 APDU format. • SW1 and SW2 of R-APDU If card is not able to respond: • SW1 = 0x64, SW2 = 0x00

<table>
<tr><th>Tag</th><th>Len</th><th>Value / Description</th></tr>
<tr><td>82</td><td>var</td><td>Encryption Control Payload tag data
• Data of R-APDU

If unencrypted:
<table>
<tr><th>Tag</th><th>Len</th><th>Value / Description</th></tr>
<tr><td>FC</td><td>var</td><td>NFC Data Container</td></tr>
<tr><td>/DF7A</td><td>var</td><td>NFC Data</td></tr>
</table>
If encrypted:
<table>
<tr><th>Tag</th><th>Len</th><th>Value / Description</th></tr>
<tr><td>/DFDF59</td><td>var</td><td>Encrypted Data Primitive to be decrypted.</td></tr>
<tr><td>/DFDF50</td><td>var</td><td>Encrypted Data KSN</td></tr>
<tr><td>/DFDF51</td><td>01</td><td>Encrypted Data Encryption Type</td></tr>
</table>
</td></tr>
</table>

Example Unencrypted Payload

```
81 0100 (Tag Respons Code)
82 82036D (Encryption Control)
   FC 820369 (NFC Data Container)
      DF7A 820364 (NFC Data)
           031391010F5504777772E6D616774656B2E636F6DFE00. . .
                         www.magtek.com
```

Example Encrypted Payload

```
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
```

4.16 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) sendNFCCommand:(IData*) data
          lastCommand: (BOOL) lastCommand
              encrypt: (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:
Returns true if 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

<table>
<tr><th>Tag</th><th>Len</th><th>Value / Description</th></tr>
<tr><td>81</td><td>01</td><td>Tag Response Code
0x00 = Success
0x01 = Failed</td></tr>
<tr><td>82</td><td>var</td><td>Encryption Control Payload

If unencrypted:
<table>
<tr><th>Tag</th><th>Len</th><th>Value / Description</th></tr>
<tr><td>FC</td><td>var</td><td>NFC Data Container</td></tr>
<tr><td>/DF7A</td><td>var</td><td>NFC Data</td></tr>
</table>

If encrypted:
<table>
<tr><th>Tag</th><th>Len</th><th>Value / Description</th></tr>
<tr><td>/DFDF59</td><td>var</td><td>Encrypted Data Primitive to be decrypted.</td></tr>
<tr><td>/DFDF50</td><td>var</td><td>Encrypted Data KSN</td></tr>
<tr><td>/DFDF51</td><td>01</td><td>Encrypted Data Encryption Type</td></tr>
</table></td></tr>
</table>

Example Unencrypted Payload

```
81 0100 (Tag Respons Code)
82 82036D (Encryption Control)
   FC 820369 (NFC Data Container)
      DF7A 820364 (NFC Data)
           031391010F5504777772E6D616774656B2E636F6DFE00. . .
                         www.magtek.com
```

Example Encrypted Payload

```
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
```

4.17 sendSelection

This function send a user selection to the device.

```
(BOOL) 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:
Returns true if successful. Otherwise, returns false.

4.18 startTransaction

This function starts a transaction. The transaction will be processed through multiple calls to the event OnEvent() or a completion handler.

`(BOOL) startTransaction:` (**IResult**
Contains status and data of operations of return type IResult.

```
@property (nonatomic) MTU_StatusCode status;
@property (nonatomic, strong) IData* data;

(IResult*) status:(MTU_StatusCode) Status data: (IData*) Data;
```

IData		
Member	Type/ Format	Description
Status	**StatusCode**	Enumerated status code.
Data	**IData***	IData

ITransaction* `transaction);`

`(void) startTransaction:` (**IResult**
Contains status and data of operations of return type IResult.

```
@property (nonatomic) MTU_StatusCode status;
@property (nonatomic, strong) IData* data;

(IResult*) status:(MTU_StatusCode) Status data: (IData*) Data;
```

IData		
Member	Type/ Format	Description
Status	**StatusCode**	Enumerated status code.

IData		
Data	**IData***	IData

ITransaction* transaction)
```
      completionHandler: (BooleanCallback) handler;
```

Parameter	Description
transaction	Instance of ITransaction.
handler	Completion handler.

Return Value:
None

4.19 subscribeAll

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

```
(Boolean) subscribeAll: (id<IEventSubscriber>) delegate;
```

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

Return Value:
Returns true if successful. Otherwise, returns false.

4.20 unsubscribeAll

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

```
(Boolean) unsubscribeAll: (id<IEventSubscriber>) delegate;
```

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

Return Value:
Returns true if successful. Otherwise, returns false.

5 DeviceCapability

Create an instance of the DeviceCapability, then use the properties described in this chapter.

5.1 AutoSignatureCapture

This property returns true if the device is capable of automatically capturing a signature during a transaction.

```
BOOL AutoSignatureCapture;
```

Return Value:
Returns true if device is capable of automatically capturing a signature. Otherwise, returns false.

5.2 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.

```
BOOL BatteryBackedClock;
```

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

5.3 Display

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

```
BOOL Display;
```

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

5.4 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.

```
BOOL MSRPowerSaver;
```

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

5.5 PaymentMethods

This property returns a bit masked option of payment methods supported by the device.

```
MTU_PaymentMethod PaymentMethods;
```

Return Value:
Returns a bit masked option of `MTU_PaymentMethod`.

5.6 PINPad

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

```
BOOL PINPad;
```

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

5.7 Signature

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

```
BOOL Signature;
```

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

5.8 SRED

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

```
BOOL SRED;
```

Return Value:
Returns 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.
- **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.

```
(Boolean) close;
```

Return Value:
Returns 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.

```
(Boolean) deviceReset;
```

Return Value:
Returns true if successful. Otherwise, returns false.

6.3 displayMessage

This function displays a predefined message on the device.

```
(Boolean) displayMessage: (Byte) messageID
                 timeout: (Byte) timeOut;

(void) displayMessage: (Byte) messageID
              timeout: (Byte) time
    completionHandler: (BooleanCallback) callback;
```

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.
callback	Completion handler.

Return Value:
Returns true if successful. Otherwise, returns false.

6.4 endSession

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

```
(Boolean) endSession;
```

Return Value:
Returns true if successful. Otherwise, returns false.

6.5 getInput

This function sends a request for user input to the device. The response data will be returned in the event OnEvent().

```
(Boolean) getInput: (IData*) data;
```

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

Return Value:
Returns true if successful. Otherwise, returns false.

6.6 open

This function opens a connection to the device.

```
(Boolean) open;
```

Return Value:
Returns true if successful. Otherwise, returns false.

6.7 playSound

This function instructs the device to play a tone.

```
(Boolean) playSound: (IData*) data;
```

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

Return Value:
Returns true if successful. Otherwise, returns false.

6.8 resetDeviceWithCompletionHandler

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. The response is sent to a completion handler.

```
(void) resetDeviceWithCompletionHandler: (BooleanCallback)callback;
```

Parameter	Description
callback	Completion handler.

Return Value:
None

6.9 send

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

```
(Boolean) 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:
Returns true if successful. Otherwise, returns false.

6.10 sendData

This function sends data.

```
(void) sendData: (IData*) data
completionHandler: (BooleanCallback) callback;
```

Parameter	Description
data	Byte array or string data to send to the device. Data must contain the full command as required by the device.
callback	Completion handler. If the operation is successful, the completion handler will return true, otherwise false. typedef void(^BooleanCallback)(Boolean resultFlag);

Return Value:
None

6.11 sendExtendedCommand

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

```
(Boolean) 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:
Returns true if successful. Otherwise, returns false.

6.12 sendSync

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

```
(IResult*) sendSync: (IData*)data;
```

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

Return Value:
Returns `IResult*`.

```
@interface IResult : NSObject

@property (nonatomic) MTU_StatusCode status;
@property (nonatomic, strong) IData* data;

+ (IResult*) status:(MTU_StatusCode) Status data: (IData*) Data;

@end
```

6.13 setDateTime

This function sets the date and time for the device.

```
(Boolean) setDateTime: (IData*) data;
```

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

Return Value:
Returns true if successful. Otherwise, returns false.

6.14 setupEADeviceProtocolString

This function sets the external accessory protocol string.

```
(void) setupEADeviceProtocolString: (NSString *)protocolString;
```

Parameter	Description
protocolString	Name of the external accessory protocol string.

Return Value:
None.

6.15 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.

```
(Boolean) setLatch: (BOOL) enableLock;
```

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

Return Value:
Returns true if successful. Otherwise, returns false.

6.16 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.

```
(Boolean) showImage: (Byte) 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:
Returns true if successful. Otherwise, returns false.

6.17 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. The response is sent to a completion handler.

```
(void) showImage: (Byte) imageID
completionHandler: (BooleanCallback) callback;
```

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.
callback	Completion handler. If the operation is successful, the completion handler will return true, otherwise false. typedef void(^BooleanCallback)(Boolean resultFlag);

Return Value:
Returns true if successful. Otherwise, returns false.

6.18 showImage

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

```
(Boolean) showImage: (ImageData*) data
            timeout: (Byte) timeOut;
```

Parameter	Description
data	ImageData object containing the image to display.
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: MTU_IMAGE_TYPE_BITMAP = BMP file
Data	NSData*	Image encoded data. Images must be BMP format, 160KB or smaller with no compression, maximum 320px by 240px, with color depth 16 color, 256 color, 16-bit color, or 24-bit color. Images smaller than the maximum size are centered on the display. Note images at full screen size must be 16-bit color or lower to meet the size requirement. For details see ***D998200383 DynaFlex Family Programmer's Manual (COMMANDS)***
BackgroundColor	NSData*	Background color in RRGGBB format. 0x000000 = Black 0xFFFFFF = White

Return Value:
Returns true if successful. Otherwise, returns false.

6.19 showBarCode

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

```
(Boolean) showBarCode: (BarCodeRequest*) request
              timeout: (Byte) timeOut
               prompt: (IData*) prompt;
```

Parameter	Description
request	BarCodeRequest object containing the barcode data to display.

Parameter	Description
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	NSData*	Data to encode into a barcode
BlockColor	NSData*	Block color. Use RRGGBB format. 0x000000 = Black
BackgroundColor	NSData*	Background color. Use RRGGBB format. 0xFFFFFF = White
ErrorCorrection	Byte	Error Correction 0x00 = Low (default) 0x01 = Medium 0x02 = Quartile 0x03 = High See **ISO/IEC 18004:2015**
MaskPattern	Byte	Mask Pattern 0x00 to 0x07 = Mask Pattern 0xFF = Device Select Optimal Mask Pattern (default) See **ISO/IEC 18004:2015**
MinVersion	Byte	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	Byte	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:
Returns true if successful. Otherwise, returns false.

6.20 showConnectedEAAccessoryIfAny

This function is to show any connected EA accessories.

```
(void) showConnectedEAAccessoryIfAny;
```

Return Value:

None

6.21 startBarCodeReader

This function starts the barcode reader. The response data will be returned in the event OnEvent().

```
(Boolean) startBarCodeReader: (Byte) timeOut
                        mode: (Byte) 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:
Returns true if successful. Otherwise, returns false.

6.22 startScan

This function scans for peripherals.

```
(void) startScan;
```

Return Value:
None

6.23 startScanBarcodeWithTimeout

This function starts the barcode reader. The response data will be returned in the event OnEvent().

```
(void) startScanBarcodeWithTimeout: (Byte) time
                    encryptionMode: (Byte) mode
                 completionHandler: (BooleanCallback) callback;
```

Parameter	Description
time	Time to enable barcode reader. Usage: 0x00 = Wait until a barcode is read or **stopBarCodeReader()** is called. 0x01 to 0xFF = 1 to 255 seconds
mode	Encrypt payload Usage: 0x00 = do not encrypt barcode data 0x01 = encrypt barcode data.
callback	Completion handler. If the operation is successful, the completion handler will return true, otherwise false. typedef void(^BooleanCallback)(Boolean resultFlag);

Return Value:
None

6.24 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.

```
(Boolean) stopBarCodeReader;
```

Return Value:
Returns true if successful. Otherwise, returns false.

6.25 stopScan

This function stops the scans for peripherals.

```
(void) stopScan;
```

Return Value:
None

6.26 stopScanBarcodeWithCompletionHandler

This function stops the barcode reader.

```
(void) startScanBarcodeWithTimeout: (BooleanCallback) callback;
```

Parameter	Description
callback	Completion handler. If the operation is successful, the completion handler will return true, otherwise false. typedef void(^BooleanCallback)(Boolean resultFlag);

Return Value:
None

6.27 turnEAAccessoryConnectionNotificationsOn

This function turns on the notifications for EA accessories.

```
(void) turnEAAccessoryConnectionNotificationsOn;
```

Return Value:
None

6.28 turnEAAccessoryConnectionNotificationsOff

This function turns off the notifications for EA accessories.

```
(void) turnEAAccessoryConnectionNotificationsOff;
```

Return Value:
None

7 ConnectionInfo

Create an instance of the ConnectionInfo, then use the function calls described in this chapter.

7.1 getAddress

This function returns the address of the device.

```
(NSString*) getAddress;
```

Return Value:
Returns the address of the device.

7.2 getConnectionType

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

```
(MTU_ConnectionType) getConnectionType;
```

Return Value:
Returns the `MTU_ConnectionType`

7.3 getDeviceType

This function returns the type for the device.

```
(MTU_DeviceType) getDeviceType;
```

Return Value:
Returns the `MTU_DeviceType.`

7.4 getCertificateInfo

This function returns the type for the device.

```
(CertificateInfo*) getCertificateInfo;
```

Return Value:
Returns the `CertificateInfo*`.

CertificateInfo:

Parameter	Type	Description
format	NSString*	Format for the certificate data. PKCS12 PFX
data	NSData*	Certificate data
password	NSString*	Password for the certificate data
ignoreNameMismatch	Boolean	Flag to ignore a mismatch between the name in the certificate and the name of the server during TLS connection authentication. false - Do not ignore a mismatched name. true - Ignore a mishmatched name.

7.5 initWithDeviceType

This function initializes an instance of ConnectionInfo.

```
(ConnectionInfo*) initWithDeviceType:( MTU_DeviceType) deviceType
                  connectionType: (MTU_ConnectionType) connectionType
                         address: (NSString*) address;

(ConnectionInfo*) initWithDeviceType: (MTU_DeviceType) deviceType
                  connectionType: (MTU_ConnectionType) connectionType
                         address: (NSString*) address
                     certificate: (CertificateInfo*) certInfo;
```

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

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>address</td><td>Address for the device.

For Ethernet devices, address should be in the form:
<code>IP://IP-Address:PORT</code>
for example, <code>IP://10.57.10.180:26</code>

For Wireless devices, address should be in the form:
<code>TLS12://TLSDEVICESERIALNUMBER</code>
<code>TLS12TRUST://TLSDEVICESERIALNUMBER</code>
for example, <code>TLS12://TLS99261829170E0810</code>
<code>TLS12TRUST://TLS99261829170E0810</code>

For Bluetooth LE devices, address should be in the form:
<code>BLEEMV://DEVICENAME</code>
for example, <code>BLEEMV://DynaPro Go-EB66</code>

For WebSocket or Secure WebSocket devices, address should be in the form:
<code>ws://IP-Address or Hostname</code>
<code>wss://IP-Address or Hostname</code>
for example, <code>ws://192.168.1.150</code>
<code>wss://b512345.magtek.com</code>

The client private key certificate (e.g. filename client.p12) must be accessible to the custom software that makes a secure WebSocket (wss://) connection to the DynaFlex II PED.

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]</td></tr>
</table>

Parameter	Description
certInfo	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*** The client private key certificate (e.g. filename client.p12) must be accessible to the custom software that makes a secure WebSocket (wss://) connection to the DynaFlex II PED. The client.p12 certificate is not needed if the connection type is not secure WebSocket (ws://)

Return Value:
Returns an IDevice*.

8 DeviceInformation

Create an instance of the **DeviceInformation** from **Idevice.getDeviceInfo**. Then use the function calls described in this chapter.

8.1 deviceModel

This property returns the model name of the device.

```
NSString* deviceModel;
```

Return Value:
Returns the model name of the device.

8.2 deviceName

This property returns the name of the device.

```
NSString* deviceName;
```

Return Value:
Returns the name of the device.

8.3 deviceSerial

This property returns the serial number of the device.

```
NSString* deviceSerial;
```

Return Value:
Returns the serial number of the device.

8.4 initWithName

This function initializes an instance of DeviceInfo.

```
(instancetype) initWithName: (NSString*) name
                      model: (NSString*) model
               serialNumber: (NSString*) sn;
```

Parameter	Description
name	Name
model	Model
sn	Serial number

9 IDeviceConfiguration

Create an instance of the IDeviceConfiguration using **IDevice.getDeviceConfiguration**. 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.
- **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) after 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.

```
(NSData*) getChallengeToken: (NSData*) data;
```

Parameter	Description
data	Byte array containing the request ID to be protected.

Return Value:
Returns a byte array containing the challenge token.

9.2 getConfigInfo

This function retrieves device configuration information. For an example, see appendix 15.20A.6 IDeviceConfiguration Walk Through.

```
(NSData*) getConfigInfo: (byte) configType
                   data: (NSData*) data;
```

Parameter	Description
configType	Type of configuration. For DynaFlex, this is the first number of the Property OID. • 0x01 = Device Settings • 0x02 = Device Information
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:
Returns an array of bytes containing the configuration information.

9.3 getDeviceInfo

This function retrieves device specific information.

```
(NSString*) getDeviceInfo: (MTU_InfoType) infoType;
```

Parameter	Description
infoType	Enumerated information type. **InfoType**

Return Value:
Returns a string value device information.

9.4 getFile

This function retrieves file information based on file ID. The response file data will be returned via the callback.

```
(int) getFile: (NSData*) fileID
     callback: (nullable id<IConfigurationCallback>) callback;
```

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

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

9.5 getKeyInfo

This function retrieves key information.

```
(NSData*) getKeyInfo: (byte) keyType
                data: (NSData*) 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:
Returns an array of bytes containing the key information.

9.6 sendFile

This function sends a file to the device.

```
(int) sendFile: (NSData*) fileID
          data: (NSData*) data
      callback: (nullable id<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.

Parameter	Description
callback	Name of a class or structure that implements the **IConfigurationCallback** Delegates events.

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

9.7 sendImage

This function sends an image to the device.

```
(int) sendImage: (byte) imageID
           data: (NSData*) data
       callback: (nullable id<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. Images must be BMP format, 160KB or smaller with no compression, maximum 320px by 240px, with color depth 16 color, 256 color, 16-bit color, or 24-bit color. Images smaller than the maximum size are centered on the display. Note images at full screen size must be 16-bit color or lower to meet the size requirement. For details see ***D998200383 DynaFlex Family Programmer's Manual (COMMANDS)***
callback	Name of a class or structure that implements the **IConfigurationCallback** Delegates events.

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

9.8 sendSecureFile

This function sends a file to the device using a secure command structure.

```
(int) sendSecureFile: (NSData*) fileID
                data: (NSData*) data
            callback: (nullable id<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 or structure that implements the **IConfigurationCallback** Delegates events.

Return Value:
Returns 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 appendix 15.20A.6 IDeviceConfiguration Walk Through.

```
(int) setConfigInfo: (byte) configType
               data: (NSData*) data
           callback: (nullable id<IConfigurationCallback>) callback;
```

Parameter	Description
configType	Type of configuration. For DynaFlex, this is the first number of the Property OID. • 0x01 = Device Settings • 0x02 = Device Information
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 or structure that implements the **IConfigurationCallback** Delegates events.

Return Value:
Returns 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) setDisplayImage: (byte) imageID;

(void) setDisplayImage: (byte) imageID
    completionHandler: (BooleanCallback) callback;
```

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.
callback	Completion handler. If the operation is successful, the completion handler will return true, otherwise false. typedef void(^BooleanCallback)(Boolean resultFlag);

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

9.11 updateFirmware

This function updates the device firmware.

```
(int) updateFirmware: (ushort) configType
                data: (NSData*) data
            callback: (nullable id<IConfigurationCallback>) callback;
```

Parameter	Description
configType	Type of firmware. For DynaFlex, use: 1 = Main App 2 = Wireless App 3 = Bluetooth LE App
data	Firmware image to be sent to the device.
callback	Name of a class or structure that implements the **IConfigurationCallback** Delegates events.

Return Value:
Returns 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: (byte) keyType
               data: (NSData*) data
           callback: (nullable id<IConfigurationCallback>) callback;
```

Parameter	Description
keyType	Type of key.
data	Key data to be sent to the device.
callback	Name of a class or structure that implements the **IConfigurationCallback** Delegates events.

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

10 ConfigurationInfo

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

10.1 configType

This property is for the configuration type.

```
Byte configType;
```

Return Value:

- 0x01 = Device Settings
- 0x02 = Device Information

10.2 OidAndValue

This property is for the Object ID and value.

```
NSData* OidAndValue;
```

Return Value:
The OID and value.

10.3 fromASN1Oid

This function is for reading and setting the device OID properties.

```
(nullable instancetype) initWithASN1Oid: (NSString*) oidPath;

(nullable instancetype) initWithASN1Oid: (NSString*) oidPath
                               hexValue: (NSString*) value;

(nullable instancetype) initWithASN1Oid: (NSString*) oidPath
                                  value: (nullable NSData*) data;
```

Parameter	Description
oidPath	Object ID path.
value	Hexadecimal string containing the value.
data	Data array containing the value.

Return Value:
OID property value.

11 Data Classes

11.1 BarCodeData

```
(instancetype) NewBarCodeData: (NSData*) data
                          ksn: (NSData*) ksn
                    encrypted: (BOOL) encrypted
                   encryption: (Byte) encryption;
```

BarCodeData		
Member	Type/ Format	Description
Data	NSData*	Data payload
Encrypted	Bool	0x00 = data is not encrypted 0x01 = data is encrypted
EncryptionType	Byte	Encryption type
KSN	NSData*	Key Serial Number
NewBarCodeData()		Initializes a BarCodeData.

11.2 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	NSString*	Certificate data format. "PKCS12" – for .p12 file. "PFX" – for .pfx file.
data	NSData*	Certificate data.
password	NSString*	Password to access the certificate data.
inithWithFormat()		Initializes CertificateInfo.

11.3 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	NSString*	String value
ByteArray	NSData*	Byte array
dataWithString()	NSString*	Initializes an IData from an NSString.
dataWithData()	NSData*	Initializes an IData from an NSData.
dataWithHex()	NSString*	Initializes an IData from an NSString formatted as hexadecimal.

Example of using IData.

```
// String usage
var data1: IData = IData.dataWithHex("3030")

// Array usage
var byteArray[2]: Byte = {0x30,0x30}
var dataBytes: NSData = NSData.dataWithBytes(byteArray, length:2)
var data2: IData = Idata.dataWithData(dataBytes)
```

11.4 IResult

Contains status and data of operations of return type IResult.

```
@property (nonatomic) MTU_StatusCode status;
@property (nonatomic, strong) IData* data;

(IResult*) status:(MTU_StatusCode) Status data: (IData*) Data;
```

IData		
Member	**Type/ Format**	**Description**
Status	**StatusCode**	Enumerated status code.
Data	**IData***	IData

11.5 ITransaction

This is the interface used as the parameter for startTransaction(). For an example, see the sample code in IDevice Walk Through.

ITransaction		
Member	**Type/ Format**	**Description**
Timeout	Byte	Transaction timeout in seconds. Default is 60 seconds. Usage: `0 to 255` - Depending on the device, 0 means no timeout.
PaymentMethods	**PaymentMethod**	List of the PaymentMethod enumeration. Usage: `MSR` - For magnetic stripe cards. `Contact` - For EMV chip cards. `Contactless` - For NFC contactless cards. `ManualEntry` - For user to manually enter transaction data without any card access. When set to ManualEntry, the other payment methods do not apply. `BarCode` - Reserved for future use. `AppleVAS` - Reserved for future use. `BarCodeEncryped` - Reserved for future use. `NFC` - For NFC MiFare. `GoogleVAS` - Reserved for future use.
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.
Amount	NSString* Max 13	EMV Tag 9F02 - Authorized amount of the transaction. Example: `"1.23"` – $1.23 `"10"` – $10.00
CashBack	NSString* Max 13	EMV Tag 9F03 - Secondary amount associated with the transaction. Example: `"1.23"` – $1.23 `"10"` – $10.00

ITransaction		
CurrenyCode	NSData* 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	NSData* 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.
DisplayAmountForQuickChip	BOOL	Display Amount for Quick Chip Transaction Flow. Usage: `false` - Do not display Amount when QuickChip mode is true. Default. `true` - Display Amount when QuickChip mode is true.

ITransaction		
OverrideFinalTransactionMessage	Byte	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	Byte	The format of the EMV response. Usage: 0x00 – Legacy. Default. 0x01 – RFU
MerchantCategory	NSData* 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	NSData* 20	EMV Tag 9F7C – Proprietary merchant data that may be requested. The byte array is null by default.
MerchantID	NSData* 15	EMV Tag 9F16 - Used to uniquely identify a given merchant. The byte array is null by default.
TransactionCategory	NSData* 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	Byte	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	Byte	Card number valid format. Usage: 0x00 – PAN min 8, max 21 digits
ManualEntrySound	Byte	Beeper feedback. Usage: 0x00 – On keypress sound disabled 0x01 – On keypress sound enabled
AppleVASMode	**VASMode**	An enumeration for the Apple VAS Mode.

ITransaction		
AppleVASProtocol	**VASProtocol**	An enumeration for the Apple VAS Protocol.

12 IEventSubscriber Delegate

MTUniversal API will invoke the callback function in this chapter to provide the requested data and/or a detailed response. To delegate the event, call the subscribeAll() function with the name of a class that implements the IEventSubscriber interface.

12.1 OnEvent

OnEvent() handles nearly all event types. The `eventType` parameter defines which event is triggered.

```
(void) OnEvent: (MTU_EventType) eventType
          Data: (IData*) data;
```

Parameter	Description
eventType	An enumeration indicating the event triggered by the device.
data	Contains the data for the event.

Return Value: None

Example:

```
class MTUSDK_DynaFlex : IEventSubscriber

func OnEvent(_ eventType: MTU_EventType, data: IData!)
{
      switch eventType
      {
         case MTU_EventType_TransactionStatus:
            break;
         case MTU_EventType_AuthorizationRequest:
            break;
         case MTU_EventType_TransactionResult:
            break;
         case MTU_EventType_PINData:
            break;
         case MTU_EventType_PANData:
            break;
         case MTU_EventType_BarCodeData:
            break
         default:
            break;
      }
}
device.subscribeAll(self)
```

13 IConfigurationCallback Delegates

This interface invokes a callback function to receive data and/or a detailed response. To use, a class must implements the **IConfigurationCallback** Delegates interface.

13.1 OnCalculateMAC

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

```
(IResult*) OnCalculateMAC: (unsigned char) macType
                     data: (NSData*) data;
```

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

Return Value:
Returns an IResult* that contains the calculated MAC.

13.2 OnProgress

This event is called to update the host on the progress of an asynchronous IDeviceConfiguration operation.

```
(void) OnProgress: (int) Progress;
```

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

Return Value: None

13.3 OnResult

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

```
(void) OnResult: (MTU_StatusCode) status
           data: (NSData*) data;
```

Parameter	Description
status	An enumerated MTU_StatusCode.
data	Contains the data for the event.

Return Value: None

Example:

```
@interface MTUSDK_DynaFlex : <IConfigurationCallback>
```

```
func OnCalculateMAC(_ macType: UInt8, data: Data) -> IResult
    {
        // Event handler
    }

func onProgress(_ progressValue: Int32)
    {
        // Event handler
    }

func onResult(_ status: MTU_StatusCode, data: Data)
    {
        // Event handler
    }
```

14 MTUSDKDelegate Delegate

MTUniversal API will invoke the callback function in this chapter to provide the requested data and/or a detailed response. To delegate the event, implements the MTUSDKDelegate interface.

14.1 onDeviceList

Handles BLE Service and EA Service events. Use this to find a BLE device or EA accessory.

```
(void) onDeviceList: (id) instance
 withConnectionType: (MTU_ConnectionType) connectionType
         deviceList: (NSArray<IDevice *> *) deviceList;
```

Parameter	Description
instance	Instance identifier.
withConnectionType	Enumerated connection type.
deviceList	Array of devices.

Return Value: None

14.2 didSystemUpdateState

Handles update of host system state.

```
(void) onDeviceList: (SystemState) state;
```

Parameter	Description
state	Enumerated system state.

Return Value: None

15 Enumerations

15.1 BarCodeFormat

This enum refers to the type of barcodes to display.

Enum	Description
MTU_BARCODE_FORMAT_BLOB	Data is binary format
MTU_BARCODE_FORMAT_COMMAND	Data is a command in binary format
MTU_BARCODE_FORMAT_BLOB_BASE64	Data is Base64 encoded format
MTU_BARCODE_FORMAT_COMMAND_BASE64	Data is a command in Base64 format

15.2 BarCodeType

This enum refers to the type of barcodes to display.

Enum	Description
MTU_QRCODE	QR code

15.3 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
MTU_ConnectionState_Unknown	Device is in an unknown connection state.
MTU_ConnectionState_Disconnected	Device is disconnected.
MTU_ConnectionState_Connecting	Device is in the process of connecting. The next state is to be Connected.
MTU_ConnectionState_Error	There was an error either connecting or disconnecting the device.
MTU_ConnectionState_Connected	Device is connected and ready for transacting.
MTU_ConnectionState_Disconnecting	Device is in the process of disconnecting. The next state will is to be Disconnected.

15.4 ConnectionType

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

Enum	Description
MTU_ConnectionType_USB (Reserved)	USB devices
MTU_ConnectionType_BLUETOOTH_LE (Reserved)	Bluetooth Low Energy devices: DynaMax
MTU_ConnectionType_BLUETOOTH_LE_EMV (Reserved)	Bluetooth Low Energy with EMV supported devices: • eDynamo
MTU_ConnectionType_BLUETOOTH_LE_EMVT (Reserved)	Bluetooth Low Energy with EMV supported devices: • tDynamo
MTU_ConnectionType_TCP (Reserved)	Transmission Control Protocol supported devices: • DynaPro
MTU_ConnectionType_TCP_TLS (Reserved)	Transmission Control Protocol with Transport Layer Security supported devices: • DynaPro Go
MTU_ConnectionType_TCP_TLS_TRUST (Reserved)	Transmission Control Protocol with Transport Layer Security supported devices: • DynaPro Go
MTU_ConnectionType_WEBSOCKET	WebSocket supported devices: • DynaFlex II PED
MTU_ConnectionType_WEBSOCKET_TRUST	WebSocket supported devices. This will establish a TLS connection to device without requirement for name match. • DynaFlex II PED
MTU_ConnectionType_SERIAL (Reserved)	UART supported devices
MTU_ConnectionType_AUDIO (Reserved)	Audio devices: • iDynamo 6
MTU_ConnectionType_EXTERNAL_ACCESSORY	iAP2 supported devices: DynaFlex II Go
MTU_ConnectionType_VIRTUAL (Reserved)	Virtual devices
MTU_ConnectionType_MQTT	MQTT devices: • DynaFlex II PED

15.5 DataEntryType

This enum is reserved for future use.

Enum	Description
MTU_DataEntry_PIN	Request Personal Identification Number
MTU_DataEntry_Signature	Request Signature
MTU_DataEntry_SSN	Request Social security number
MTU_DataEntry_ZIPCODE	Request Zip code
MTU_DataEntry_BirthDate	Request Birth date
MTU_DataEntry_ActivationCode	Request Activation code

15.6 DeviceEvent

This enum refers to a change in the device status.

Enum	Description
MTU_DeviceEvent_None	No event to report.
MTU_DeviceEvent_DeviceResetOccured	A device reset had occurred.
MTU_DeviceEvent_DeviceResetWillOccur	A device reset will occur soon. Host application may use this as a warning to take appropriate actions.
MTU_DeviceEvent_DeviceNotPaired	Device is not paired therefore cannot be connected.

15.7 DeviceFeature

This enum refers to a featured supported by the device.

Enum	Description
MTU_DeviceFeature_None	No feature
MTU_DeviceFeature_SignatureCapture	Supports signature capture
MTU_DeviceFeature_PINEntry	Supports PIN entry
MTU_DeviceFeature_PANEntry	Supports PAN entry
MTU_DeviceFeature_ShowBarCode	Supports display of a barcode
MTU_DeviceFeature_ScanBarCode	Supports scanning a barcode

15.8 DeviceType

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

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

15.9 EventType

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

Enum	Description
MTU_EventType_ConnectionState	There was a change in the connection state of the device.
MTU_EventType_DeviceResponse	Device has responded to a command.
MTU_EventType_DeviceExtendedResponse	Device has responded to an extended command.
MTU_EventType_DeviceNotification	Device has sent a notification.
MTU_EventType_DeviceTransferCancelled	Device has cancelled data transfer.

Enum	Description
MTU_EventType_CardData	Device has sent magnetic stripe data from a card swipe.
MTU_EventType_TransactionStatus	There was a change in transaction status.
MTU_EventType_DisplayMessage	Device has a message to display for the user.
MTU_EventType_InputRequest	Device is requesting input from the user.
MTU_EventType_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.
MTU_EventType_TransactionResult	Device has sent the result of the transaction.
MTU_EventType_PINBlock	Device has sent the PINBlock after the user has entered a PIN on the device.
MTU_EventType_Signature	Device has sent data which represents a signature from a user.
MTU_EventType_DeviceDataFile	Device has sent a data file.
MTU_EventType_OperationStatus	Device has sent an operation status of a command.
MTU_EventType_DeviceEvent	Device has sent change of device state.
MTU_EventType_UserEvent	Device has sent a notification related to user interaction with the device.
MTU_EventType_FeatureStatus	Device has sent a change in a feature.
MTU_EventType_PINData	Device has sent data related to a PIN.
MTU_EventType_PANData	Device has sent data related to a PAN.
MTU_EventType_BarCodeData	Device has sent barcode data.
MTU_EventType_NFCEvent	Device has sent NFC event.
MTU_EventType_NFCData	Device has sent NFC data.
MTU_EventType_NFCResponse	Device has sent response to an NFC command.
MTU_EventType_NFCRAPDUResponse	Device has sent response to an NFC APDU command for Mifare DESFire Tag.
MTU_EventType_ClearDisplay	Device has sent a notification to clear the display of messages for the user.
MTU_EventType_EnhancedInputRequest	Device is requesting enhanced input from the user.

15.10 FeatureStatus

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

Enum	Description
MTU_FeatureStatus_NoStatus	No change in status
MTU_FeatureStatus_Success	Success
MTU_FeatureStatus_Failed	Failed
MTU_FeatureStatus_TimedOut	Timed out
MTU_FeatureStatus_Cancelled	Cancelled
MTU_FeatureStatus_Error	Error
MTU_FeatureStatus_HardwareNA	Featured hardware not applicable for a status

15.11 ImageType

This enum refers to the type of image.

Enum	Description
MTU_IMAGE_TYPE_BITMAP	BMP file

15.12 InfoType

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

Enum	Description
MTU_InfoType_DeviceSerialNumber	Device serial number.
MTU_InfoType_FirmwareVersion	Firmware version of the device.
MTU_InfoType_DeviceCapabilities	Capabilities of the device delimited by a comma.
MTU_InfoType_Boot1Version	Boot 1 firmware version of the device.
MTU_InfoType_Boot0Version	Boot 0 firmware version of the device.
MTU_InfoType_FirmwareHash	Firmware hash comprised of part numbers, versions, and timestamps.
MTU_InfoType_TamperStatus	Tamper status of the device. 0x00 = Not Tampered 0x01 = Tampered
MTU_InfoType_OperationStatus	Operation status of the device. 0x01 = Offline 0x02 = Online

Enum	Description
MTU_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
MTU_InfoType_DeviceModel	Device model.

15.13 PaymentMethod

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

Enum	Description
MTU_PaymentMethod_MSR	For magnetic stripe cards.
MTU_PaymentMethod_Contact	For EMV chip cards.
MTU_PaymentMethod_Contactless	For NFC contactless cards.
MTU_PaymentMethod_ManualEntry	For user to manually enter transaction data without any card access.
MTU_PaymentMethod_BarCode	Reserved for future use.
MTU_PaymentMethod_AppleVAS	For Apple VAS.
MTU_PaymentMethod_BarCodeEncrypted	Reserved for future use.
MTU_PaymentMethod_NFC	For NFC Mifare.
MTU_PaymentMethod_GoogleVAS	Reserved for future use.

15.14 StatusCode

This enum refers to the status of the transaction.

Enum	Description
MTU_StatusCode_Success	The operation completed successfully.
MTU_StatusCode_Timeout	The operation timed out.
MTU_StatusCode_Error	Error attempting the operation.
MTU_StatusCode_Unavailable	Status currently unavailable.

15.15 SystemState

This enum refers to the host system state of connectivity.

Enum	Description
SystemStateUnknown	Initial state
SystemStateBluetoothLEResetting	Bluetooth resetting
SystemStateBluetoothLEUnsupported	Bluetooth unsupported
SystemStateBluetoothLEUnauthorized	Bluetooth unauthorized
SystemStateBluetoothLEPoweredOff	Bluetooth powered off
SystemStateBluetoothLEPoweredOn	Bluetooth powered on
SystemStateNetworkOff	Network off
SystemStateNetworkOn	Network on
SystemStateServerNotReachable	Server not reachable
SystemStateRejectedByServer	Rejected by server
SystemStateTLSAuthenticationFailed	TLS authentication failed
SystemStateProtocolError	Protocol error
SystemStateServerConnected	Server connected

15.16 TransactionStatus

This enum refers to the status of the transaction.

Enum	Description
MTU_TransactionStatus_NoStatus	Set before the start of a transaction and before a card is presented to the device.
MTU_TransactionStatus_NoTransaction	No transaction in progress.
MTU_TransactionStatus_CardSwiped	A card was swiped into the device.
MTU_TransactionStatus_CardInserted	A card was inserted into the device.
MTU_TransactionStatus_CardRemoved	A card was removed from the device.
MTU_TransactionStatus_CardDetected	A card was detected by the device.
MTU_TransactionStatus_CardCollision	A card collision was detected by the device. Possibly more than 1 contactless detected.
MTU_TransactionStatus_TimedOut	The transaction was not completed before a timeout period.
MTU_TransactionStatus_HostCanceled	The host software sent a cancel.

Enum	Description
MTU_TransactionStatus_TransactionCanceled	The transaction was canceled by device.
MTU_TransactionStatus_TransactionInProgress	The transactions is in progress.
MTU_TransactionStatus_TransactionError	There is an error during the transaction.
MTU_TransactionStatus_TransactionApproved	The transactions is approved.
MTU_TransactionStatus_TransactionDeclined	The transactions is declined.
MTU_TransactionStatus_TransactionCompleted	The transaction is completed.
MTU_TransactionStatus_TransactionFailed	The transaction failed.
MTU_TransactionStatus_TransactionNotAccepted	The transaction was not accepted by the device.
MTU_TransactionStatus_SignatureCaptureRequested	A signature capture is requested by the device.
MTU_TransactionStatus_TechnicalFallback	Due to technical reasons, the chip transaction cannot be completed by the reader.
MTU_TransactionStatus_QuickChipDeferred	Device has sent a “Z3” response code to the chip card.
MTU_TransactionStatus_DataEntered	Data has been entered on the device for a manual card entry transaction.
MTU_TransactionStatus_TryAnotherInterface	Due to removal of the chip card, error with contactless card, or magnetic swipe has a service code for chip card, the transaction cannot be completed by the reader.
MTU_TransactionStatus_BarcodeRead	A barcode is read.
MTU_TransactionStatus_VASError	Apple VAS error occurred.
MTU_TransactionStatus_MSRFallback	MSR fallback occurred.
MTU_TransactionStatus_ReservedICCOnly	

15.17 OperationStatus

This enum refers to the operation status of the device.

Enum	Description
MTU_OperationStatus_NoStatus	No update for the operation.
MTU_OperationStatus_Started	Device has started an operation.
MTU_OperationStatus_Warning	Device has sent a warning about the operation.
MTU_OperationStatus_Failed	Device has failed an operation.
MTU_OperationStatus_Done	Device has completed an operation.

15.18 UserEvent

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

Enum	Description
MTU_UserEvent_None	No events yet to occur.
MTU_UserEvent_ContactlessCardPresented	Contactless card has been presented.
MTU_UserEvent_ContactlessCardRemoved	Contactless card has been removed.
MTU_UserEvent_CardSeated	Card is seated into the chip station.
MTU_UserEvent_CardUnseated	Card was removed from the chip station.
MTU_UserEvent_CardSwiped	Magnetic stripe card was swiped.
MTU_UserEvent_TouchPresented	Touch screen sensor press detected.
MTU_UserEvent_TouchRemoved	Touch screen sensor release detected.
MTU_UserEvent_BarcodeRead	Barcode detected.
MTU_UserEvent_NFCMifareUltralightPresented	Mifare Ultralight presented.
MTU_UserEvent_MifareClassic1KPresented	Mifare Classic 1K presented.
MTU_UserEvent_MifareClassic4KPresented	Mifare Classic 4K presented.
MTU_UserEvent_MifareDESFirePresented	Mifare DESFire Light presented.
MTU_UserEvent_NFCMifareUltralightRemoved	Mifare Ultralight removed.
MTU_UserEvent_MifareClassic1KRemoved	Mifare Classic 1K removed.
MTU_UserEvent_MifareClassic4KRemoved	Mifare Classic 4K removed.
MTU_UserEvent_MifareDESFireRemoved	Mifare DESFire Light removed.

15.19 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
MTU_VASMode_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.
MTU_VASMode_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.
MTU_VASMode_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.

15.20 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.createDevice() → IDevice
- CoreAPI.setDeviceType()
- CoreAPI.startScanningForPeripherals → onDeviceList()

These examples demonstrate methods for creating an IDevice to be used in the MagTek Universal SDK. This also shows how to establish a device specific API, which is not used with the MagTek Universal SDK.

Here, a single IDevice is established.

```
// Access MMS with Universal SDK using createDevice()

var mtmms: IDevice = CoreAPI.createDevice(
      MTU_DeviceType_MMS,
      connection: MTU_ConnectionType_ WEBSOCKET,
      address: "ws://b123456.magtek.com",
      model: "DynaFlex",
      name: webSocketAddress,
      serial: "B123456",
      cert: Certificate)
```

BLE device.

```
// Acess BLE device with Universal SDK

private let mtusdkShared = CoreAPI.shared()
mtusdkShared.setDeviceType(MTU_DeviceType_MMS, andConnectionType:
MTU_ConnectionType_BLUETOOTH_LE_EMV)

mtusdkShared.mtuSDKDelegate = self
mtusdkShared.startScanningForPeripherals()
```

```
// Acess onDeviceList()

func onDeviceList(_ instance: Any, withConnectionType:
MTU_ConnectionType, deviceList: [Any])
{
        if let devices = deviceList as? [IDevice] {
            deviceAddressList = []
        }
}
```

iAP2 Lightning device.

```
// Acess iAP2 device with Universal SDK

private let mtusdkShared = CoreAPI.shared()
mtusdkShared.setDeviceType(MTU_DeviceType_MMS, andConnectionType:
MTU_ConnectionType_EXTERNAL_ACCESSORY)

mtusdkShared.setupEADeviceProtocolString(MTUConstant.dynaFlex2GoProtoc
olString)
mtusdkShared.mtuSDKDelegate = self
mtusdkShared.turnEAAccessoryConnectionNotificationsOn()
mtusdkShared.showConnectedEAAccessoryIfAny()
```

```
// Acess onDeviceList()

func onDeviceList(_ instance: Any, withConnectionType:
MTU_ConnectionType, deviceList: [Any])
{
        if let devices = deviceList as? [IDevice] {
            deviceAddressList = []
        }
}
```

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

```
// Extend the main window to receive events.
@interface MTUSCK_DynaFlex : <IEventSubscriber,
IConfigurationCallback>

// Establish a device from CoreAPI.
var device: IDevice = CoreAPI.createDevice(
      MTU_DeviceType_MMS,
      connection:MTU_ConnectionType_ WEBSOCKET,
      address:"ws://b123456.magtek.com",
      "",
      "DynaFlex",
      "",
      cert:Certificate)


/* 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. */
var result: Bool = device.subscribeAll(self)

// Assign parameters for the transaction.
var transaction: ITransaction = ITransaction(
Amount:“1.00”,
CashBack:“0.00”,
TransactionType:0,
Timeout:30,
For:MTU_PaymentMethod_MSR | MTU_PaymentMethod_Contact |
MTU_PaymentMethod_Contactless,
QuickChip:true
)

// Start transaction.
result = device.startTransaction(transaction)
```

A.2.1 Handling Events

Application Main window may extent the IEventSubscriber. This example demonstrates how to parse for the various event types.

Example

```
@interface MTUSCK_DynaFlex : <IEventSubscriber,
IConfigurationCallback>

func onEvent(_ eventType: MTU_EventType, data: IData!)
{
    switch eventType
    {
```

Various events are separately shown below.

```
case MTU_EventType_ConnectionState:
// Parse for the ConnectionState
var value: MTU_ConnectionState = ConnectionStateBuilder.GetValue(data)

break;
```

```
case MTU_EventType_DeviceResponse:

break;
```

```
case MTU_EventType_DeviceExtendedResponse:

break;
```

```
case MTU_EventType_DeviceNotification:

break;
```

```
case MTU_EventType_CardData:

break;
```

```
case MTU_EventType_DisplayMessage:

// Get the message.
var message: NSString = data.StringValue

break;
```

```
case MTU_EventType_InputRequest:

break;
```

```
case MTU_EventType_AuthorizationRequest:

// Forward ARQC to processor.
```

```
/* data[0..1] – ARQC length
   data[2..n] – remainder contains the ARQC TLV object
*/

var ARQC: IData = IData.dataWithData(data.ByteArray)

// App function to send the request to the processor.
var ARPC: NSString =
sendARQCToProcessorForApproval(ARQC.data.ByteArray);

// Send authorization to device when not in QuickChip mode.
var ARPCTLV: IData = IData.alloc(init);
ARPCTLV.StringValue = "FF7413DFDF250742363243413546FA067004” + ARPC;

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

break;
```

```
case MTU_EventType_TransactionResult:

/* data[0]    – Signature Required
   data[1..2] – Batch Data length
   data[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[0].

break;
```

```
case EventType.PINData:

break;
```

```
case MTU_EventType_Signature:

break;
```

```
case MTU_EventType_TransactionStatus:

/*
Transaction status enumeration is build from
the TransactionStatusBuilder.

*/
```

```
var status: MTU_TransactionStatus = TransactionStatusBuilder.
GetStatusCode(data.StringValue);

if status == MTU_TransactionStatus_CardSwiped
{
  //
}
if status == MTU_TransactionStatus_CardInserted
{
  //
}

if status == MTU_TransactionStatus_TransactionApproved
{
  //
}

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 deviceControl from device.
var deviceControl: IDeviceControl = device.getDeviceControl

// Open the device, then use the IDeviceControl functions.
var result: BOOL = deviceControl.open()

. . .

// Close the device.
result = deviceControl.close()
```

A.4 ConnectionInfo Walk Through

The following walks through how to make use of ConnectionInfo.

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

Example

```
// Establish a ConnectionInfo from device.
var connectionInfo: ConnectionInfo = device.getConnectionInfo()

// Retrieve address, connectionType, and deviceType.
var address: NSString = connectionInfo.getAddress()
var connectionType: MTU_ConnectionType = connectionInfo
.getConnectionType()
var deviceType: MTU_DeviceType = connectionInfo.getDeviceType()
```

A.5 IDeviceCapability Walk Through

The following walks through how to make use of IDeviceCapability.

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

```
// Establish a IDeviceCapabilities from device.
DeviceCapabilities* capabilities = device.getDeviceCapability


// Retrieve device capabilities.
BOOL batteryBackedClock = capabilities.BatteryBackedClock()
if (batteryBackedClock)
{
  // Call IDeviceControl setDateTime
}

// Retrieve supported card payment methods.
MTU_PaymentMethod paymentMethods = capabilities.PaymentMethods

. . .
```

A.6 IDeviceConfiguration Walk Through

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

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

Example

```
IDeviceConfiguration* devConfig = device.getDeviceConfiguration

IDeviceControl* devControl = device.getDeviceControl
result = devControl.open()

// Update firmware.
var error: IData*
var fileData: NSData = NSData.dataWithContentsOfFile(
"/Users/UserName/DynaFlex/fw.bin",
options: NSDataReadingUncached,
error: &error);

var return: Int = devConfig.updateFirmware(1, data: fileData,
callback: self)


/* 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.
*/
var configType: Byte = 1;
var str: NSString = "E208E106E104E102C100"
var data: NSData= HexUtil.getBytesFromHexString(str)
var response: NSData = devConfig.getConfigInfo(configType, data)


/* 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.
*/
configType = 1
str = "E209E107E105E103C10101"
data NSData = HexUtil.getBytesFromHexString(str)
return = devConfig.setConfigInfo(configType, data, callback:self)
```

A.6.1 Handling Events

Application Main window may extent the **IConfigurationCallback** Delegates. This example demonstrates how to parse for the various events.

Example

```
func onProgress(_ progressValue: Int32)
{

}
```

```
func onResult(_ status: MTU_StatusCode, data: Data)
{
     /* Handle result.
     A configuration process is complete when
     status = MTU_StatusCode_Success */
}
```

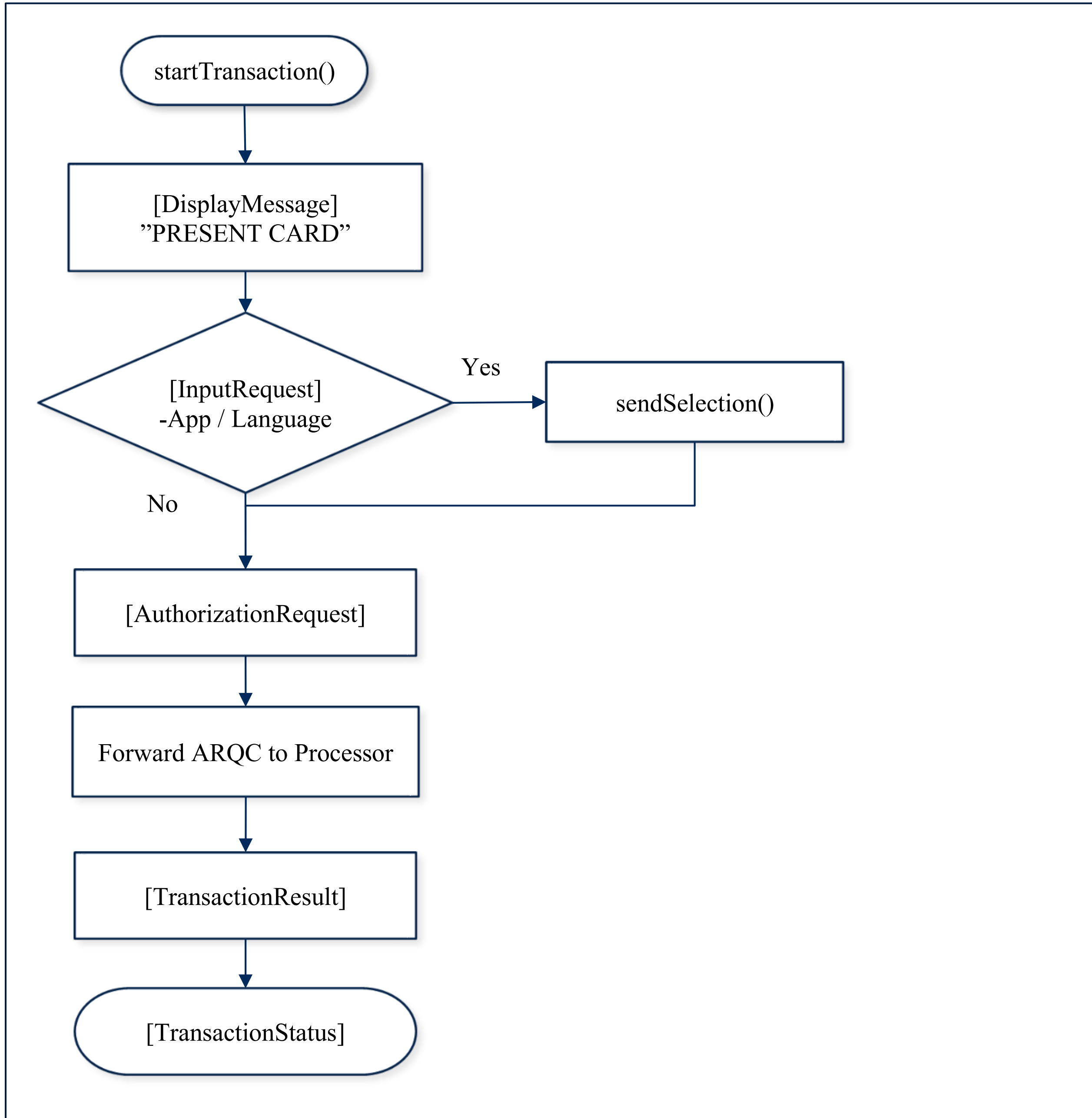
```
func OnCalculateMAC (_ macType: UInt8, data: Data) -> IResult
{
      //Calculate MAC.

}
```

Appendix B EMV Transaction Flow

This section demonstrates transaction flow.

B.1 Flow Chart - QuickChip



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.
var transaction: ITransaction = ITransaction(
Amount:"1.00",
CashBack:"0.00",
TransactionType:0,
Timeout:30,
For:MTU_PaymentMethod_MSR | MTU_PaymentMethod_Contact |
MTU_PaymentMethod_Contactless,
QuickChip:true)

// Start transaction.
var result: BOOL = device.startTransaction(transaction)
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var message: NSString = ""
  switch eventType
    {
      case MTU_EventType_DisplayMessage:
        // Get the message.
        message = data.StringValue
    }
}
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var message: NSString = ""
  switch eventType
    {
      case MTU_EventType_InputRequest:
        // Get the message.
        message = data.StringValue

        // display/retrieve user selection.

        // set status and selection result.
        var statusSelection: NSString* = "0001"
        var selectionData: IData = IData.dataWithHex(statusSelection);
        device.sendSelection(selectionData)
    }
}
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
```

```
{
  var ARQC[]: Byte = nil;
  switch eventType
    {
      case MTU_EventType_AuthorizationRequest:
        // Forward ARQC to processor.
        /* data[0..1] – ARQC length
           data[2..n] – remainder contains the ARQC TLV object */
    }
}
```

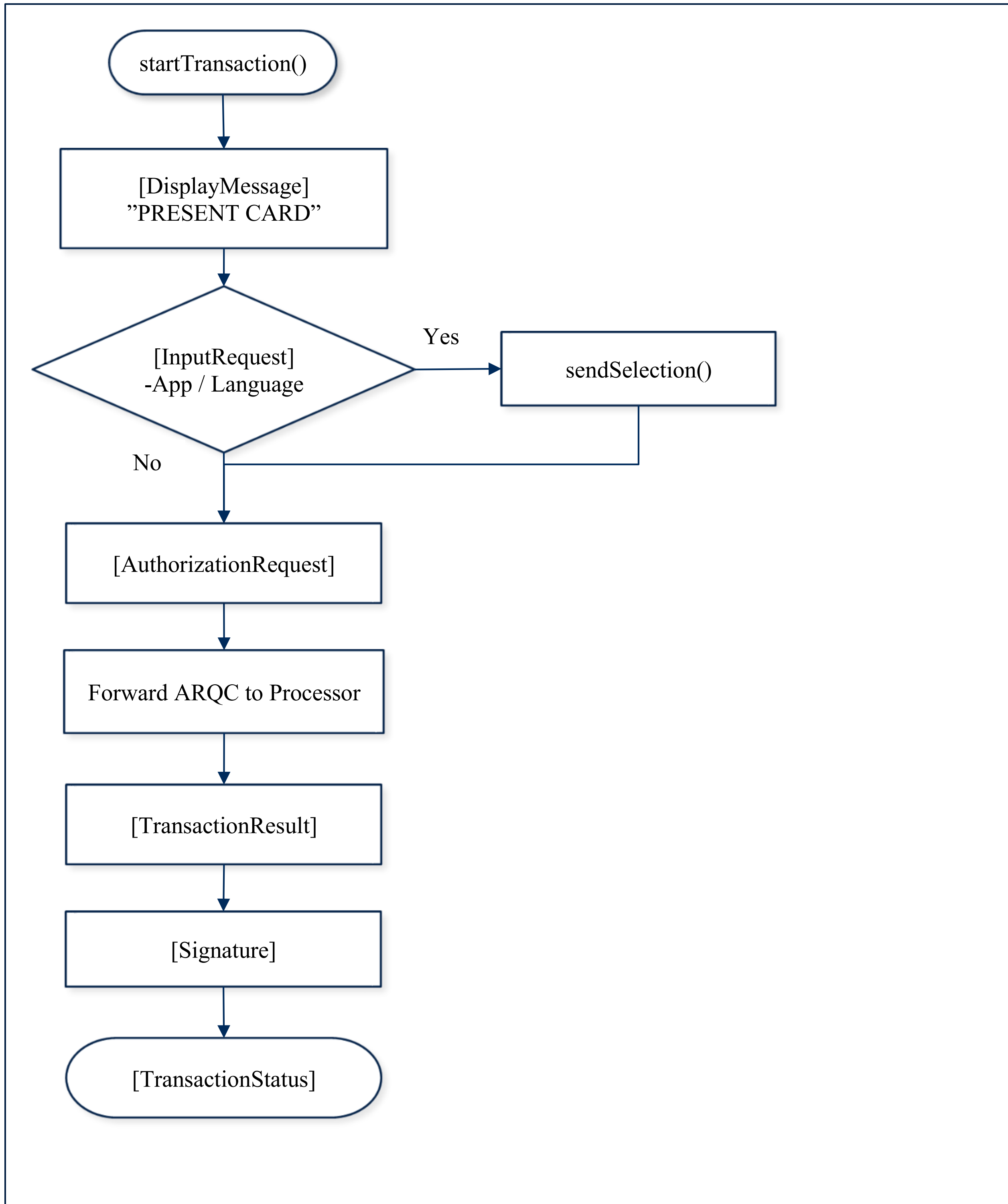
```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var message: NSString = “”
  switch eventType
    {
      case MTU_EventType_DisplayMessage:
        // Display approval message.
        message = data.StringValue

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

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  switch eventType
    {
      case MTU_EventType_TransactionResult:
         /* data[0]    – Signature Required
            data[1..2] – Batch Data length
            data[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 Flow Chart – Signature Capture



B.4 Sample Code – Signature Capture

The following breaks out the EMV flow chart into code. Events are shown separately and in the order received.

```
// Assign parameters.
var transaction: ITransaction = ITransaction(
Amount: "1.00",
CashBack: "0.00",
TransactionType:0,
Timeout:30,
For:MTU_PaymentMethod_MSR | MTU_PaymentMethod_Contact |
MTU_PaymentMethod_Contactless,
QuickChip:true
)

// Start transaction.
var result: BOOL = device.startTransaction(transaction)
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var message: NSString = ""
  switch eventType
    {
      case MTU_EventType_DisplayMessage:
        // Get the message.
        message = data.StringValue
  }
}
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var message: NSString = ""
  switch eventType
    {
      case MTU_EventType_InputRequest:
        // Get the message.
        message = data.StringValue

        // display/retrieve user selection.

        // set status and selection result.
        var statusSelection: NSString* = "0001"
        IData* selectionData = IData dataWithHex(statusSelection);
        device.sendSelection(selectionData);
    }
}
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
```

```
  var ARQC[]: Byte = nil;
  switch eventType
    {
      case MTU_EventType_AuthorizationRequest:
        // Forward ARQC to processor.
        /* data[0..1] – ARQC length
           data[2..n] – remainder contains the ARQC TLV object */
    }
}
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var message: NSString = ""
  switch eventType
    {
      case MTU_EventType_DisplayMessage:
        // Display approval message.
        message = data.StringValue;

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

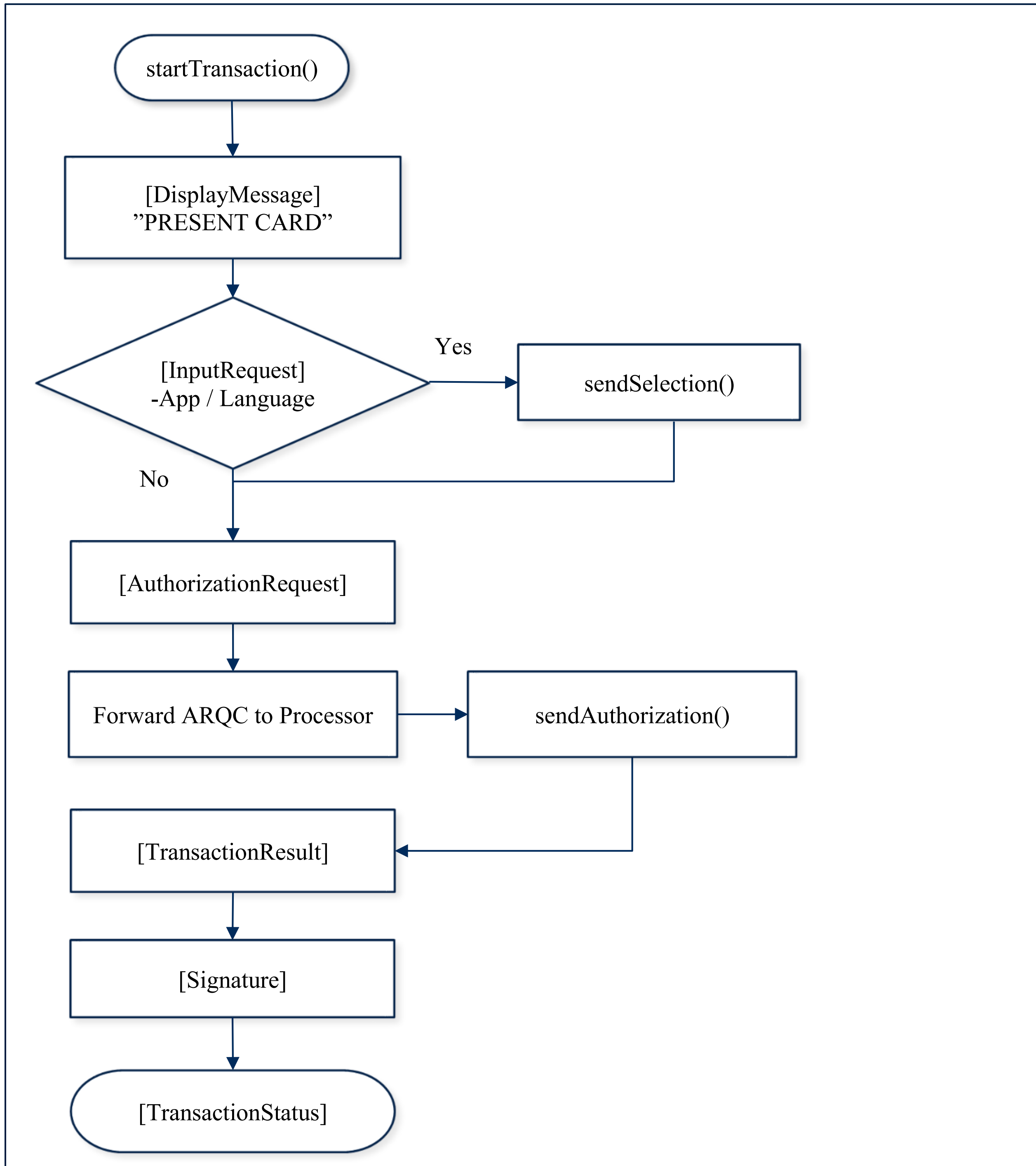
```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  switch eventType
    {
      case MTU_EventType_TransactionResult:
         /* data[0]    – Signature Required
            data[1..2] – Batch Data length
            data[3..n] – remainder contains the Batch Data TLV object
         */

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

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var signature: NSString = "";
  switch eventType
    {
      case MTU_EventType_Signature:
```

```
            signature = data.StringValue;
      }
}
```

B.5 Flow Chart – With ARPC



B.6 Sample Code – With ARPC

The following breaks out the EMV flow chart into code. When disabling QuickChip mode, host must send the ARPC to the device to complete the transaction. Events are shown separately and in the order received.

```
// Assign parameters.
var transaction: ITransaction = ITransaction(
Amount:"1.00",
CashBack:"0.00",
TransactionType:0,
Timeout:30,
For:MTU_PaymentMethod_MSR | MTU_PaymentMethod_Contact |
MTU_PaymentMethod_Contactless,
QuickChip:false
);

// Start transaction.
var result: BOOL = device.startTransaction(transaction)
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var message: NSString = ""
  switch eventType
    {
      case MTU_EventType_DisplayMessage:
        // Get the message.
        message = data.StringValue
  }
}
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var message: NSString = "";
  switch eventType
    {
      case MTU_EventType_InputRequest:
        // Get the message.
        message = data.StringValue

        // display/retrieve user selection.

        // set status and selection result.
        var statusSelection: NSString = "0001"
        var selectionData: IData* = IData
dataWithHex(statusSelection);
        device.sendSelection(selectionData)
    }
}
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var ARQC: IData = data;
  switch eventType
    {
      case MTU_EventType_AuthorizationRequest:
        // Forward ARQC to processor.
        /* data[0..1] - ARQC length
           data[2..n] - remainder contains the ARQC TLV object */
        ARPC = sendARQCToProcessorForApproval(ARQC.ByteArray)
    }
}
```

After the ARPC is returned from the processor, it is constructed into a TLV container and then sent to the device. The ARPC for approved (00) is set in ASCII 3030.

The optional tags 91, 71, and 72 (Issuer Authentication Data, Issuer Script Template 1, and Issuer Script Template 2) are not included in this example.

See the construction of the ARPCTLV in the table below.

```
var ARPC: NSString = “8A3030”;
var ARPCTLV: IData = IData();
ARPCTLV.StringValue = "FF7413DFDF25074236324341354 6FA067004” + ARPC

device.sendAuthorization(ARPCTLV)
```

ARPC TLV object for sendAuthorization().

Tag	Len	Value / Description	Typ	Req	Default
FF74	var	Container for non-MAC ARPC	T	R	
/DFDF25	var	Device Serial Number (IFD Serial Number)	B	R	
/FA	var	Container for generic data	T	R	
//70	var	Container for ARPC	T	R	
///8A	02	Authorization Response Code • 0x3030 = Approved • 0x3031 = Issuer Referral • 0x3035 = Declined • 0x3132 = Switch Interface • 0x3133 = Request Online PIN	AN	R	
///91	var	Issuer Authentication Data As defined in ***EMV Integrated Circuit Card Specifications for Payment Systems 4.3***	B	O	

Tag	Len	Value / Description	Typ	Req	Default
///71	var	Issuer Script Template 1 As defined in ***EMV Integrated Circuit Card Specifications for Payment Systems 4.3***. The host may include as many instances of this parameter as needed, up to a maximum length of 128 bytes including Tags and Lengths.	B	O	
///72	var	Issuer Script Template 2 As defined in ***EMV Integrated Circuit Card Specifications for Payment Systems 4.3***. The host may include as many instances of this parameter as needed, up to a maximum length of 128 bytes including Tags and Lengths.	B	O	

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  var message: NSString = ""
  switch eventType
    {
      case MTU_EventType_DisplayMessage:
        // Display approval message.
        message = data.StringValue

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

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
  switch eventType
    {
      case MTU_EventType_TransactionResult:
         /* data[0]    - Signature Required
            data[1..2] - Batch Data length
            data[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].
    }
}
```

```
func onEvent(_ eventType: MTU_EventType, data: IData!)
{
```

```
  var signature: NSString* = ""
  switch eventType
    {
      case MTU_EventType_Signature:
         signature = data.StringValue
    }
}
```

B.7 MSR Fallback Flow

The use case for an MSR fallback is when communication with the chip results in a terminated transaction and the `TransactionStatus` is reported as `MSRFallback`.

The host application will re-attempt the transaction. To invoke this use case, here are the following pre-requisites.

Pre-requisites:

- Device already configured for Device-Driven Fallback = Disabled.
- A card to cause the fallback. Example but not limited to a card with no applications programmed or a card with an application not configured on the device.

Scheme:

- →Host begins an initial transaction with `PaymentMethod` set to `MSR+Chip+Contactless`.
- ←Device responds with fail and with status of `MSRFallback`.
- →Host displays a message to use magnetic stripe.
- →Host starts a transaction with `PaymentMethod` set to `MSR`.
- ←Device may respond with transaction canceled card read error.
- →Host displays a message each time the transaction fails until successful or until Host decides to end the transaction.
- ←Device sends the transaction result.