# DTDevices

Generated by Doxygen 1.8.3.1

Thu Feb 28 2013 09:43:53

# Contents

# Chapter 1

# Module Documentation

## 1.1 Linea SDK

Provides access to Linea device series.

### Modules

- Delegate Notifications

  *Notifications sent by the sdk on various events - barcode scanned, magnetic card data, communication status, etc.*
- General functions

  *Functions to connect/disconnect, set delegate, make sounds, update firmware, control various device settings.*
- Magnetic Stripe Reader Functions

  *Functions to work with the Linea's built-in magenetic card reader.*
- Barcode Reader Functions

  *Functions for scanning barcodes, various barcode settings and direct control of the barcode engine.*
- Cryptographic & Security Functions

  *Starting from firmware 2.13, Linea provides strong cryptographic support for magnetic card data.*
- Bluetooth Functions

  *Functions to work with Linea's built-in bluetooth module.*
- External Serial Port Functions

  *Functions to work with Linea Tab's external serial port.*
- Encrypted Magnetic Head Functions

  *Functions to work with Linea's optional encrypted magnetic head.*
- RF Reader Functions

  *Functions to work with the Linea's built-in RF cards reader.*

### Classes

- class DTRFCardInfo

  *Information about RF card. More...*
- protocol <LineaDelegate>

  *Protocol describing various notifications that LineaSDK can send. More...*
- class Linea

  *Provides access to Linea functions. More...*

## Macros

- #define **LINEA_NO_EXCEPTIONS**
- #define **buttonPressed** lineaButtonPressed
- #define **buttonReleased** lineaButtonReleased
- #define **btmSetEnabled** btSetEnabled
- #define **btmGetEnabled** btGetEnabled
- #define **btmWrite** btWrite
- #define **btmRead** btRead
- #define **btmReadLine** btReadLine
- #define **btmGetLocalName** btGetLocalName
- #define **prnDiscoverPrinters** btDiscoverPrinters
- #define **prnDiscoverPrintersInBackground** btDiscoverPrintersInBackground
- #define **barcodeEngineSetInitString** barcodeOpticonSetInitString
- #define **msStartScan** msEnable
- #define **msStopScan** msDisable
- #define **setMSCardDataMode** msSetCardDataMode
- #define **getMSCardDataMode** msGetCardDataMode
- #define **startScan** barcodeStartScan
- #define **stopScan** barcodeStopScan
- #define **getScanTimeout** barcodeGetScanTimeout
- #define **setScanTimeout** barcodeSetScanTimeout
- #define **getScanButtonMode** barcodeGetScanButtonMode
- #define **setScanButtonMode** barcodeSetScanButtonMode
- #define **setScanBeep** barcodeSetScanBeep
- #define **getScanMode** barcodeGetScanMode
- #define **setScanMode** barcodeSetScanMode
- #define **enableBarcode** barcodeEnableBarcode
- #define **isBarcodeEnabled** barcodeIsBarcodeEnabled
- #define **isBarcodeSupported** barcodeIsBarcodeSupported
- #define **getBarcodeTypeMode** barcodeGetTypeMode
- #define **setBarcodeTypeMode** barcodeSetTypeMode
- #define **cryptoRawAuthenticateLinea** cryptoRawAuthenticateDevice
- #define **cryptoRawAuthenticateiPod** cryptoRawAuthenticateHost
- #define **cryptoAuthenticateLinea** cryptoAuthenticateDevice
- #define **cryptoAuthenticateiPod** cryptoAuthenticateHost
- #define **__has_feature**(x) 0
- #define **STRUCTURES_DEFINED**
- #define ALG_AES256 0

  *AES 256 encryption algorithm.*
- #define ALG_EH_ECC 1

  *Encrypted Head ECC encryption algorithm.*
- #define ALG_EH_AES256 2

  *Encrypted Head AES 256 encryption algorithm.*
- #define ALG_EH_IDTECH 3

  *Encrypted Head IDTECH encryption algorithm.*
- #define ALG_EH_MAGTEK 4

  *Encrypted Head MAGTEK encryption algorithm.*
- #define ALG_EH_3DES 5

  *Encrypted Head 3DES encryption algorithm.*
- #define ALG_EH_RSA_OAEP 6

  *Encrypted Head RSA encryption algorithm.*
- #define KEY_AUTHENTICATION 0x00

*Authentication key.*

- #define KEY_ENCRYPTION 0x01

  *Encryption key, if set magnetic card data will be encrypted.*

- #define KEY_EH_AES256_LOADING 0x02

  *Encrypted head key loading key.*

- #define KEY_EH_TMK_AES 0x10

  *Encrypted head TMK key.*

- #define KEY_EH_DUKPT_MASTER 0x20

  *Encrypted head DUKPT master key.*

- #define KEY_AUTH_FLAG_LOCK 1

  *This flag locks barcode, magnetic card and bluetooth usage, so it will be possible to use them only after authenticating.*

## Enumerations

- enum **BARCODES** {
  **BAR_ALL** =0, **BAR_UPC**, **BAR_CODABAR**, **BAR_CODE25_NI2OF5**,
  **BAR_CODE25_I2OF5**, **BAR_CODE39**, **BAR_CODE93**, **BAR_CODE128**,
  **BAR_CODE11**, **BAR_CPCBINARY**, **BAR_DUN14**, **BAR_EAN2**,
  **BAR_EAN5**, **BAR_EAN8**, **BAR_EAN13**, **BAR_EAN128**,
  **BAR_GS1DATABAR**, **BAR_ITF14**, **BAR_LATENT_IMAGE**, **BAR_PHARMACODE**,
  **BAR_PLANET**, **BAR_POSTNET**, **BAR_INTELLIGENT_MAIL**, **BAR_MSI**,
  **BAR_POSTBAR**, **BAR_RM4SCC**, **BAR_TELEPEN**, **BAR_PLESSEY**,
  **BAR_PDF417**, **BAR_MICROPDF417**, **BAR_DATAMATRIX**, **BAR_AZTEK**,
  **BAR_QRCODE**, **BAR_MAXICODE**, **BAR_LAST** }
- enum **BARCODES_EX** {
  **BAR_EX_ALL** =0, **BAR_EX_UPCA**, **BAR_EX_CODABAR**, **BAR_EX_CODE25_NI2OF5**,
  **BAR_EX_CODE25_I2OF5**, **BAR_EX_CODE39**, **BAR_EX_CODE93**, **BAR_EX_CODE128**,
  **BAR_EX_CODE11**, **BAR_EX_CPCBINARY**, **BAR_EX_DUN14**, **BAR_EX_EAN2**,
  **BAR_EX_EAN5**, **BAR_EX_EAN8**, **BAR_EX_EAN13**, **BAR_EX_EAN128**,
  **BAR_EX_GS1DATABAR**, **BAR_EX_ITF14**, **BAR_EX_LATENT_IMAGE**, **BAR_EX_PHARMACODE**,
  **BAR_EX_PLANET**, **BAR_EX_POSTNET**, **BAR_EX_INTELLIGENT_MAIL**, **BAR_EX_MSI_PLESSEY**,
  **BAR_EX_POSTBAR**, **BAR_EX_RM4SCC**, **BAR_EX_TELEPEN**, **BAR_EX_UK_PLESSEY**,
  **BAR_EX_PDF417**, **BAR_EX_MICROPDF417**, **BAR_EX_DATAMATRIX**, **BAR_EX_AZTEK**,
  **BAR_EX_QRCODE**, **BAR_EX_MAXICODE**, **BAR_EX_RESERVED1**, **BAR_EX_RESERVED2**,
  **BAR_EX_RESERVED3**, **BAR_EX_RESERVED4**, **BAR_EX_RESERVED5**, **BAR_EX_UPCA_2**,
  **BAR_EX_UPCA_5**, **BAR_EX_UPCE**, **BAR_EX_UPCE_2**, **BAR_EX_UPCE_5**,
  **BAR_EX_EAN13_2**, **BAR_EX_EAN13_5**, **BAR_EX_EAN8_2**, **BAR_EX_EAN8_5**,
  **BAR_EX_CODE39_FULL**, **BAR_EX_ITA_PHARMA**, **BAR_EX_CODABAR_ABC**, **BAR_EX_CODABAR_-CX**,
  **BAR_EX_SCODE**, **BAR_EX_MATRIX_2OF5**, **BAR_EX_IATA**, **BAR_EX_KOREAN_POSTAL**,
  **BAR_EX_CCA**, **BAR_EX_CCB**, **BAR_EX_CCC**, **BAR_EX_LAST** }
- enum CONN_STATES { CONN_DISCONNECTED =0, CONN_CONNECTING, CONN_CONNECTED }

  *Connection state.*

- enum BLUETOOTH_FILTER { BLUETOOTH_FILTER_ALL =-1, BLUETOOTH_FILTER_PRINTERS =1, BLUETOOTH_FILTER_PINPADS =2, BLUETOOTH_FILTER_BARCODE_SCANNERS =4 }

  *Filtering bluetooth devices to discover.*

- enum SCAN_MODES {
  MODE_SINGLE_SCAN =0, MODE_MULTI_SCAN, MODE_MOTION_DETECT, MODE_SINGLE_SCAN_RELEASE,
  MODE_MULTI_SCAN_NO_DUPLICATES }

  *Barcode scan modes.*

- enum BUTTON_STATES { BUTTON_DISABLED =0, BUTTON_ENABLED }

  *Button modes.*

- enum MS_MODES { MS_PROCESSED_CARD_DATA =0, MS_RAW_CARD_DATA }

*Card data mode.*

- enum BT_MODES { BARCODE_TYPE_DEFAULT =0, BARCODE_TYPE_EXTENDED, BARCODE_TYPE_ISO15424 }

  *The way to return barcode types.*

- enum UPDATE_PHASES { UPDATE_INIT =0, UPDATE_ERASE, UPDATE_WRITE, UPDATE_FINISH, UPDATE_COMPLETING }

  *Firmware update phases.*

- enum RF_CARD_TYPES { CARD_UNKNOWN =0, CARD_MIFARE_MINI, CARD_MIFARE_CLASSIC_1K, CARD_MIFARE_CLASSIC_4K, CARD_MIFARE_ULTRALIGHT, CARD_MIFARE_ULTRALIGHT_C, CARD_ISO14443A, CARD_MIFARE_PLUS, CARD_ISO15693, CARD_MIFARE_DESFIRE, CARD_ISO14443B, CARD_FELICA }

  *RF card types.*

- enum FELICA_SMARTTAG_BATERY_STATUSES { FELICA_SMARTTAG_BATTERY_NORMAL1 =0, FELICA_SMARTTAG_BATTERY_NORMAL2, FELICA_SMARTTAG_BATTERY_LOW1, FELICA_SMARTTAG_BATTERY_LOW2 }

  *FeliCa SmartTag battery status.*

### Variables

- const uint8_t KEY_AES256_EMPTY [32]

  *In the case where the AES256 key can be disabled to return the devce to plain text (LP without encrypted head), loading this key will remove it.*

- NSString ∗const InfoDeviceName

  *Device name as string, for example "Linea".*

- NSString ∗const InfoDeviceModel

  *Device model, if any, for example "XAMBL".*

- NSString ∗const InfoFirmwareRevision

  *Firmware revision as string, for example 2.41.*

- NSString ∗const InfoFirmwareRevisionNumber

  *Firmware revision as number, useful for comparison, for example 241.*

## 1.1.1 Detailed Description

Provides access to Linea device series. In order to use Linea in your program, several steps have to be performed:

- Include LineaSDK.h and libdtdev.a in your project.
- Go to Frameworks and add ExternalAccessory framework
- Edit your program plist file, add new element and select "Supported external accessory protocols" from the list, then add two items to it - com.datecs.linea.pro.msr and com.datecs.linea.pro.bar

## 1.1.2 Class Documentation

### 1.1.2.1 class DTRFCardInfo

Information about RF card.

Inherits NSObject.

#### Properties

- int type

  *RF card type, one of the CARD_∗ constants.*
- NSString ∗ typeStr

  *RF card type as string, useful for display purposes.*
- NSData ∗ UID

  *RF card unique identifier, if any.*
- int ATQA

  *Mifare card ATQA.*
- int SAK

  *Mifare card SAK.*
- int AFI

  *ISO15693 card AFI.*
- int DSFID

  *ISO15693 card DSFID.*
- int blockSize

  *ISO15693 card block size.*
- int nBlocks

  *ISO15693 card number of blocks.*

### 1.1.2.2 protocol LineaDelegate-p

Protocol describing various notifications that LineaSDK can send.

#### Instance Methods

- (void) - connectionState:

  *Notifies about the current connection state.*
- (void) - buttonPressed:

  *Notification sent when some of the Linea's buttons is pressed.*
- (void) - buttonReleased:

  *Notification sent when some of the Linea's buttons is released.*
- (void) - barcodeData:type:

  *Notification sent when barcode is successfuly read.*
- (void) - barcodeData:isotype:

  *Notification sent when barcode is successfuly read.*
- (void) - magneticCardData:track2:track3:

  *Notification sent when magnetic card is successfuly read.*
- (void) - magneticCardRawData:

  *Notification sent when magnetic card is successfuly read.*
- (void) - magneticCardEncryptedData:data:

  *Notification sent when magnetic card is successfuly read.*
- (void) - magneticCardEncryptedData:tracks:data:

  *Notification sent when magnetic card is successfuly read.*
- (void) - magneticCardEncryptedData:tracks:data:track1masked:track2masked:track3:

  *Notification sent when magnetic card is successfuly read.*
- (void) - magneticCardEncryptedRawData:data:

  *Notification sent when magnetic card is successfuly read.*
- (void) - firmwareUpdateProgress:percent:

*Notification sent when firmware update process advances.*

- (void) - bluetoothDiscoverComplete:

  *Notification sent when bluetooth discovery finds new bluetooth device.*
- (void) - bluetoothDeviceDiscovered:name:

  *Notification sent when bluetooth discovery finds new bluetooth device.*
- (void) - magneticJISCardData:

  *Notification sent when JIS I & II magnetic card is successfuly read.*
- (void) - rfCardDetected:info:

  *Notification sent when a new supported RFID card enters the field.*
- (void) - rfCardRemoved:

  *Notification sent when the card leaves the field.*

#### 1.1.2.3 class Linea

Provides access to Linea functions.

Inherits NSObject.

##### Instance Methods

- (void) - addDelegate:

  *Allows unlimited delegates to be added to a single class instance.*
- (void) - removeDelegate:

  *Removes delegate, previously added with addDelegate.*
- (void) - connect

  *Tries to connect to Linea in the background, connection status notifications will be passed through the delegate.*
- (void) - disconnect

  *Stops the sdk from trying to connect to Linea and breaks existing connection.*
- (BOOL) - **isPresent**
- (BOOL) - getBatteryCapacity:voltage:error:

  *Returns Linea's battery capacity.*
- (BOOL) - playSound:beepData:length:error:

  *Makes Linea plays a sound using the built-in speaker.*
- (BOOL) - getCharging:error:

  *Returns if Linea is charging the iOS device from it's own battery.*
- (BOOL) - setCharging:error:

  *Enables or disables Lines's capability to charge the handheld from it's own battery.*
- (NSDictionary ∗) - getFirmwareFileInformation:error:

  *Returns information about the specified firmware data.*
- (BOOL) - updateFirmwareData:error:

  *Updates Linea's firmware with specified firmware data.*
- (BOOL) - getSyncButtonMode:error:

  *Returns the current sync button mode.*
- (BOOL) - setSyncButtonMode:error:

  *Sets Linea's sync button mode.*
- (BOOL) - msEnable:

  *Enables reading of magnetic cards.*
- (BOOL) - msDisable:

  *Disables magnetic card scanning started with msEnable.*
- (NSDictionary ∗) - msProcessFinancialCard:track2:

*Helper function to parse financial card and extract the data - name, number, expiration date.*

- (BOOL) - msGetCardDataMode:error:

  *Returns the current magnetic card data mode.*

- (BOOL) - msSetCardDataMode:error:

  *Sets Linea's magnetic card data mode.*

- (NSString ∗) - barcodeType2Text:

  *Helper function to return string name of barcode type.*

- (BOOL) - barcodeStartScan:

  *Starts barcode engine.*

- (BOOL) - barcodeStopScan:

  *Stops ongoing scan started with barcodeStartScan.*

- (BOOL) - barcodeGetScanTimeout:error:

  *Returns the current scan timeout.*

- (BOOL) - barcodeSetScanTimeout:error:

  *Sets the scan timeout.*

- (BOOL) - barcodeGetScanButtonMode:error:

  *Returns the current scan button mode.*

- (BOOL) - barcodeSetScanButtonMode:error:

  *Sets Linea's scan button mode.*

- (BOOL) - barcodeSetScanBeep:volume:beepData:length:error:

  *Sets Linea's beep, which is used upon successful barcode scan.*

- (BOOL) - barcodeGetScanMode:error:

  *Returns the current scan mode.*

- (BOOL) - barcodeSetScanMode:error:

  *Sets Linea's barcode engine scan mode.*

- (BOOL) - barcodeEnableBarcode:enabled:error:

  *Enables or disables reading of specific barcode type.*

- (BOOL) - barcodeIsBarcodeEnabled:

  *Returns if the the engine is set to read the barcode type or not.*

- (BOOL) - barcodeIsBarcodeSupported:

  *Returns if the the engine can read the barcode type or not.*

- (BOOL) - barcodeGetTypeMode:error:

  *Returns the current barcode type mode.*

- (BOOL) - barcodeSetTypeMode:error:

  *Sets barcode type mode.*

- (BOOL) - barcodeEnginePowerControl:error:

  *Allows basic control over the power to the barcode engine.*

- (BOOL) - barcodeEnginePowerControl:maxTimeMinutes:error:

  *Allows basic control over the power to the barcode engine.*

- (BOOL) - barcodeEngineResetToDefaults:

  *Performs factory reset of the barcode module.*

- (BOOL) - barcodeOpticonSetInitString:error:

  *Allows for a custom initialization string to be sent to the Opticon barcode engine.*

- (BOOL) - barcodeOpticonSetParams:saveToFlash:error:

  *Sends configuration parameters directly to the opticon barcode engine.*

- (NSString ∗) - barcodeOpticonGetIdent:

  *Reads barcode engine's identification.*

- (BOOL) - barcodeOpticonUpdateFirmware:bootLoader:error:

  *Performs firmware update on the optiocon 2D barcode engines.*

- (BOOL) - barcodeCodeSetParam:value:error:

  *Sends configuration parameters directly to the code barcode engine.*

- (BOOL) - barcodeCodeGetParam:value:error:

  *Reads configuration parameters directly from the code barcode engine.*
- (BOOL) - barcodeCodeUpdateFirmware:data:error:

  *Performs firmware update on the Code 2D barcode engines.*
- (NSDictionary ∗) - **barcodeCodeGetInformation:**
- (BOOL) - barcodeIntermecSetInitData:error:

  *Allows for a custom initialization string to be sent to the Intermec barcode engine.*
- (NSData ∗) - cryptoRawGenerateRandomData:

  *Generates 16 byte block of random numbers, required for some of the other crypto functions.*
- (BOOL) - cryptoRawSetKey:encryptedData:keyVersion:keyFlags:error:
- (BOOL) - cryptoSetKey:key:oldKey:keyVersion:keyFlags:error:

  *Used to store AES256 keys into Linea internal memory.*
- (BOOL) - cryptoGetKeyVersion:keyVersion:error:

  *Returns key version.*
- (NSData ∗) - cryptoRawAuthenticateDevice:error:
- (BOOL) - cryptoAuthenticateDevice:error:
- (BOOL) - cryptoRawAuthenticateHost:error:
- (BOOL) - cryptoAuthenticateHost:error:
- (BOOL) - btGetEnabled:error:

  *Returns bluethooth module status.*
- (BOOL) - btSetEnabled:error:

  *Enables or disables bluetooth module.*
- (BOOL) - btWrite:length:error:

  *Sends data to the connected remote device.*
- (BOOL) - btWrite:error:

  *Sends data to the connected remote device.*
- (int) - btRead:length:timeout:error:

  *Tries to read data from the connected remote device for specified timeout.*
- (NSString ∗) - btReadLine:error:

  *Tries to read string data, ending with CR/LF up to specifed timeout.*
- (NSString ∗) - btGetLocalName:

  *Retrieves local bluetooth name, this is the name that Linea will report to bluetooth discovery requests.*
- (NSArray ∗) - btDiscoverDevices:maxTime:codTypes:error:

  *Performs synchronous discovery of the nearby bluetooth devices.*
- (NSString ∗) - btGetDeviceName:error:

  *Queries device name given the address, this function complements the btDiscoverDevices/btDiscoverPrinters and as such is not recommended, use btDiscoverDevicesInBackground instead.*
- (BOOL) - btDiscoverDevicesInBackground:maxTime:codTypes:error:

  *Performs background discovery of nearby bluetooth devices.*
- (BOOL) - btDiscoverSupportedDevicesInBackground:maxTime:filter:error:

  *Performs background discovery of nearby supported bluetooth devices.*
- (NSArray ∗) - btDiscoverPrinters:maxTime:error:

  *Performs discovery of supported printers.*
- (BOOL) - btDiscoverPrintersInBackground:maxTime:error:

  *Performs background discovery of supported printers.*
- (BOOL) - btDiscoverPrintersInBackground:

  *Performs background discovery of supported printers.*
- (NSArray ∗) - btDiscoverPinpads:maxTime:error:

  *Performs discovery of supported pinpads.*
- (BOOL) - btDiscoverPinpadsInBackground:maxTime:error:

  *Performs background discovery of supported printers.*

- (BOOL) - btDiscoverPinpadsInBackground:

  *Performs background discovery of supported printers.*
- (BOOL) - btConnect:pin:error:

  *Tries to connect to remote device.*
- (BOOL) - btDisconnect:error:

  *Disconnects from remote device.*
- (BOOL) - btEnableWriteCaching:error:

  *Enables or disables write caching on the bluetooth stream.*
- (BOOL) - btSetDataNotificationMaxTime:maxLength:sequenceData:error:

  *Sets the conditions to fire the NSStreamEventHasBytesAvailable event on bluetooth streams.*
- (BOOL) - extOpenSerialPort:baudRate:parity:dataBits:stopBits:flowControl:error:

  *Opens the external serial port with specified settings.*
- (BOOL) - extCloseSerialPort:error:

  *Closes the external serial port opened with extOpenSerialPort.*
- (BOOL) - extWriteSerialPort:data:error:

  *Sends data to the connected remote device via serial port.*
- (NSData ∗) - extReadSerialPort:length:timeout:error:

  *Reads data from the connected remote device via serial port.*
- (NSDictionary ∗) - emsrGetFirmwareInformation:error:

  *Returns information about the specified head firmware data.*
- (BOOL) - emsrIsTampered:error:

  *Checks if the head was tampered or not.*
- (BOOL) - emsrGetKeyVersion:keyVersion:error:

  *Retrieves the key version (if any) of a loaded key.*
- (BOOL) - emsrLoadInitialKey:error:

  *Loads Terminal Master Key (TMK) or reenable after tampering.*
- (BOOL) - emsrLoadKey:error:

  *Loads new key, in plain or encrypted with already loaded AES256 Key Encryption Key (KEK).*
- (NSData ∗) - emsrGetDUKPTSerial:

  *Returns DUKPT serial number, if DUKPT key is set.*
- (NSString ∗) - emsrGetDeviceModel:

  *Returns head's model.*
- (BOOL) - emsrGetFirmwareVersion:error:

  *Returns head's firmware version as number MAJOR∗100+MINOR, i.e.*
- (BOOL) - emsrGetSecurityVersion:error:

  *Returns head's security version as number MAJOR∗100+MINOR, i.e.*
- (NSData ∗) - emsrGetSerialNumber:

  *Return head's unique serial number as byte array.*
- (BOOL) - emsrUpdateFirmware:error:

  *Performs firmware update on the encrypted head.*
- (NSArray ∗) - emsrGetSupportedEncryptions:

  *Returns supported encryption algorhtms by the encrypted head.*
- (BOOL) - emsrSetEncryption:params:error:

  *Selects the prefered encryption algorithm.*
- (BOOL) - emsrConfigMaskedDataShowExpiration:unmaskedDigitsAtStart:unmaskedDigitsAtEnd:error:

  *Fine-tunes which part of the card data will be masked, and which will be sent in clear text for display/print purposes.*
- (BOOL) - rfInit:error:

  *Initializes and powers on the RF card reader module.*
- (BOOL) - rfClose:

  *Powers down RF card reader module.*
- (BOOL) - rfRemoveCard:error:

*Call this function once you are done with the card, a delegate call rfCardRemoved will be called when the card leaves the RF field and new card is ready to be detected.*

- (BOOL) - mfAuthByKey:type:address:key:error:

  *Authenticate mifare card block with direct key data.*
- (BOOL) - mfStoreKeyIndex:type:key:error:

  *Store key in the internal module memory for later use.*
- (BOOL) - mfAuthByStoredKey:type:address:keyIndex:error:

  *Authenticate mifare card block with previously stored key.*
- (NSData ∗) - mfRead:address:length:error:

  *Reads one more more blocks of data from Mifare Classic/Ultralight cards.*
- (int) - mfWrite:address:data:error:

  *Writes one more more blocks of data to Mifare Classic/Ultralight cards.*
- (BOOL) - mfUlcSetKey:key:error:

  *Sets the 3DES key of Mifare Ultralight C cards.*
- (BOOL) - mfUlcAuthByKey:key:error:

  *Performs 3DES authentication of Mifare Ultralight C card using the given key.*
- (NSData ∗) - iso15693Read:startBlock:length:error:

  *Reads one more more blocks of data from ISO 15693 card.*
- (int) - iso15693Write:startBlock:data:error:

  *Writes one more more blocks of data to ISO 15693 card.*
- (NSData ∗) - iso15693GetBlocksSecurityStatus:startBlock:nBlocks:error:

  *Reads the security status of one more more blocks from ISO 15693 card.*
- (BOOL) - iso15693LockBlock:block:error:

  *Locks a single ISO 15693 card block.*
- (BOOL) - iso15693WriteAFI:afi:error:

  *Changes ISO 15693 card AFI.*
- (BOOL) - iso15693LockAFI:error:

  *Locks ISO 15693 AFI preventing further changes.*
- (BOOL) - iso15693WriteDSFID:dsfid:error:

  *Changes ISO 15693 card DSFID.*
- (BOOL) - iso15693LockDSFID:error:

  *Locks ISO 15693 card DSFID preventing further changes.*
- (NSData ∗) - felicaRead:startBlock:length:error:

  *Reads one more more blocks of data from FeliCa card.*
- (int) - felicaWrite:startBlock:data:error:

  *Writes one more more blocks of data to FeliCa card.*
- (BOOL) - felicaSmartTagGetBatteryStatus:status:error:

  *Returns FeliCa SmartTag battery status.*
- (BOOL) - felicaSmartTagClearScreen:error:

  *Clears the screen of FeliCa SmartTag.*
- (BOOL) - felicaSmartTagDrawImage:image:topLeftX:topLeftY:drawMode:layout:error:

  *Draws image on FeliCa SmartTag's screen.*
- (BOOL) - felicaSmartTagSaveLayout:layout:error:

  *Saves the current display as layout number.*
- (BOOL) - felicaSmartTagDisplayLayout:layout:error:

  *Displays previously stored layout.*
- (int) - felicaSmartTagWrite:address:data:error:

  *Writes data in FeliCa SmartTag.*
- (NSData ∗) - felicaSmartTagRead:address:length:error:

  *Writes data in FeliCa SmartTag.*
- (BOOL) - felicaSmartTagWaitCompletion:error:

  *Waits for FeliCa SmartTag to complete current operation.*

#### Class Methods

- (id) + sharedDevice

  *Creates and initializes new Linea class instance or returns already initalized one.*

#### Properties

- NSInputStream * btInputStream

  *Bluetooth input stream, you can use it after connecting with btConnect.*
- NSOutputStream * btOutputStream

  *Bluetooth output stream, you can use it after connecting with btConnect.*
- id delegate

  *Adds delegate to the class.*
- NSMutableArray * delegates

  *Provides a list of currently registered delegates.*
- int connstate

  *Returns current connection state.*
- NSString * deviceName

  *Returns connected device name.*
- NSString * deviceModel

  *Returns connected device model.*
- NSString * firmwareRevision

  *Returns connected device firmware version.*
- NSString * hardwareRevision

  *Returns connected device hardware version.*
- NSString * serialNumber

  *Returns connected device serial number.*
- int sdkVersion

  *SDK version number in format MAJOR*100+MINOR, i.e.*

##### 1.1.2.3.1 Property Documentation

###### 1.1.2.3.1.1 - (int) sdkVersion `[read],[atomic],[assign]`

SDK version number in format MAJOR*100+MINOR, i.e.

version 1.15 will be returned as 115

### 1.1.3 Enumeration Type Documentation

#### 1.1.3.1 enum BLUETOOTH_FILTER

Filtering bluetooth devices to discover.

#### Enumerator

***BLUETOOTH_FILTER_ALL*** Include all supported devices (default)

***BLUETOOTH_FILTER_PRINTERS*** Include supported printers.

***BLUETOOTH_FILTER_PINPADS*** Include supported pinpads.

***BLUETOOTH_FILTER_BARCODE_SCANNERS*** Include supported barcode scanners.

#### 1.1.3.2 enum BT_MODES

The way to return barcode types.

**Enumerator**

***BARCODE_TYPE_DEFAULT*** Barcode types are returned from the BAR_∗ list (default)

***BARCODE_TYPE_EXTENDED*** Barcode types are returned from the extended barcode list - BAR_EX_∗.

***BARCODE_TYPE_ISO15424*** Barcode types are returned as ISO 15424 format.

#### 1.1.3.3 enum BUTTON_STATES

Button modes.

**Enumerator**

***BUTTON_DISABLED*** Button is disabled.

***BUTTON_ENABLED*** Button is enabled (default)

#### 1.1.3.4 enum CONN_STATES

Connection state.

**Enumerator**

***CONN_DISCONNECTED*** Device is disconnected, no automatic connection attempts will be made.

***CONN_CONNECTING*** The SDK is trying to connect to the device.

***CONN_CONNECTED*** Device is connected.

#### 1.1.3.5 enum FELICA_SMARTTAG_BATERY_STATUSES

FeliCa SmartTag battery status.

**Enumerator**

***FELICA_SMARTTAG_BATTERY_NORMAL1*** Normal, card can be used.

***FELICA_SMARTTAG_BATTERY_NORMAL2*** Normal, card can be used.

***FELICA_SMARTTAG_BATTERY_LOW1*** Low, consider replacing.

***FELICA_SMARTTAG_BATTERY_LOW2*** Very Low, replace it.

#### 1.1.3.6 enum MS_MODES

Card data mode.

**Enumerator**

***MS_PROCESSED_CARD_DATA*** Card data is processed and tracks are extracted (default)

***MS_RAW_CARD_DATA*** Card data will be returned as RAW sequence of bits.

#### 1.1.3.7 enum RF_CARD_TYPES

RF card types.

**Enumerator**

***CARD_UNKNOWN*** Unknown card.

***CARD_MIFARE_MINI*** Mifare Mini.

***CARD_MIFARE_CLASSIC_1K*** Mifare Classic 1K.

***CARD_MIFARE_CLASSIC_4K*** Mifare Classic 4K.

***CARD_MIFARE_ULTRALIGHT*** Mifare Ultralight.

***CARD_MIFARE_ULTRALIGHT_C*** Mifare Ultralight C.

***CARD_ISO14443A*** ISO 14443A.

***CARD_MIFARE_PLUS*** Mifare Plus.

***CARD_ISO15693*** ISO 15693.

***CARD_MIFARE_DESFIRE*** Mifare Desfire.

***CARD_ISO14443B*** ISO 14443B.

***CARD_FELICA*** FeliCa.

#### 1.1.3.8 enum SCAN_MODES

Barcode scan modes.

**Enumerator**

***MODE_SINGLE_SCAN*** The scan will be terminated after successful barcode recognition (default)

***MODE_MULTI_SCAN*** Scanning will continue unless either scan button is releasd, or stop scan function is called.

***MODE_MOTION_DETECT*** For as long as scan button is pressed or stop scan is not called the engine will operate in low power scan mode trying to detect objects entering the area, then will turn on the lights and try to read the barcode. Supported only on Code engine.

***MODE_SINGLE_SCAN_RELEASE*** Pressing the button/start scan will enter aim mode, while a barcode scan will actually be performed upon button release/stop scan.

***MODE_MULTI_SCAN_NO_DUPLICATES*** Same as multi scan mode, but allowing no duplicate barcodes to be scanned.

#### 1.1.3.9 enum UPDATE_PHASES

Firmware update phases.

**Enumerator**

***UPDATE_INIT*** Initializing update.

***UPDATE_ERASE*** Erasing old firmware/preparing memory.

***UPDATE_WRITE*** Writing data.

***UPDATE_FINISH*** Update complete, this is the final phase.

***UPDATE_COMPLETING*** Post-update operations.

## 1.2 Delegate Notifications

Notifications sent by the sdk on various events - barcode scanned, magnetic card data, communication status, etc.

### Functions

- (void) - <LineaDelegate>::connectionState:

  *Notifies about the current connection state.*
- (void) - <LineaDelegate>::buttonPressed:

  *Notification sent when some of the Linea's buttons is pressed.*
- (void) - <LineaDelegate>::buttonReleased:

  *Notification sent when some of the Linea's buttons is released.*
- (void) - <LineaDelegate>::barcodeData:type:

  *Notification sent when barcode is successfuly read.*
- (void) - <LineaDelegate>::barcodeData:isotype:

  *Notification sent when barcode is successfuly read.*
- (void) - <LineaDelegate>::magneticCardData:track2:track3:

  *Notification sent when magnetic card is successfuly read.*
- (void) - <LineaDelegate>::magneticCardRawData:

  *Notification sent when magnetic card is successfuly read.*
- (void) - <LineaDelegate>::magneticCardEncryptedData:data:

  *Notification sent when magnetic card is successfuly read.*
- (void) - <LineaDelegate>::magneticCardEncryptedData:tracks:data:

  *Notification sent when magnetic card is successfuly read.*
- (void) - <LineaDelegate>::magneticCardEncryptedData:tracks:data:track1masked:track2masked:track3:

  *Notification sent when magnetic card is successfuly read.*
- (void) - <LineaDelegate>::magneticCardEncryptedRawData:data:

  *Notification sent when magnetic card is successfuly read.*
- (void) - <LineaDelegate>::firmwareUpdateProgress:percent:

  *Notification sent when firmware update process advances.*
- (void) - <LineaDelegate>::bluetoothDiscoverComplete:

  *Notification sent when bluetooth discovery finds new bluetooth device.*
- (void) - <LineaDelegate>::bluetoothDeviceDiscovered:name:

  *Notification sent when bluetooth discovery finds new bluetooth device.*
- (void) - <LineaDelegate>::magneticJISCardData:

  *Notification sent when JIS I & II magnetic card is successfuly read.*
- (void) - <LineaDelegate>::rfCardDetected:info:

  *Notification sent when a new supported RFID card enters the field.*
- (void) - <LineaDelegate>::rfCardRemoved:

  *Notification sent when the card leaves the field.*

### 1.2.1 Detailed Description

Notifications sent by the sdk on various events - barcode scanned, magnetic card data, communication status, etc.

### 1.2.2 Function Documentation

#### 1.2.2.1 - (void) barcodeData: (NSString ∗) *barcode* isotype:(NSString ∗) *isotype*

Notification sent when barcode is successfuly read.

This notification is used when barcode type is set to BARCODE_TYPE_ISO15424, or barcode engine is CR-800.

**Parameters**

| | |
|---|---|
| *barcode* | - string containing barcode data |
| *type* | - barcode type, one of the BAR_∗ constants |

#### 1.2.2.2 - (void) barcodeData: (NSString ∗) *barcode* type:(int) *type*

Notification sent when barcode is successfuly read.

This notification is used when barcode type is set to BARCODE_TYPE_DEFAULT or BARCODE_TYPE_EXTENDED.

**Parameters**

| | |
|---|---|
| *barcode* | - string containing barcode data |
| *type* | - barcode type, one of the BAR_∗ constants |

#### 1.2.2.3 - (void) bluetoothDeviceDiscovered: (NSString ∗) *btAddress* name:(NSString ∗) *btName*

Notification sent when bluetooth discovery finds new bluetooth device.

**Parameters**

| | |
|---|---|
| *btAddress* | bluetooth address of the device |
| *btName* | bluetooth name of the device |

#### 1.2.2.4 - (void) bluetoothDiscoverComplete: (BOOL) *success*

Notification sent when bluetooth discovery finds new bluetooth device.

**Parameters**

| | |
|---|---|
| *success* | true if the discovery complete successfully, even if it not resulted in any device found, false if there was an error communicating with the bluetooth module |

#### 1.2.2.5 - (void) buttonPressed: (int) *which*

Notification sent when some of the Linea's buttons is pressed.

**Parameters**

| | | |
|---|---|---|
| *which* | button identifier, one of: | |
| | 0 | right scan button |

#### 1.2.2.6 - (void) buttonReleased: (int) *which*

Notification sent when some of the Linea's buttons is released.

**Parameters**

| | |
|---|---|
| *which* | button identifier, one of: |

| | |
|---|---|
| 0 | right scan button |

#### 1.2.2.7 - (void) connectionState: (int) *state*

Notifies about the current connection state.

**Parameters**

| | |
|---|---|
| *state* | - connection state, one of: |

| | |
|---|---|
| CONN_DISCONNECTED | there is no connection to Linea and the sdk will not try to make one even if the device is attached |
| CONN_CONNECTING | Linea is not currently connected, but the sdk is actively trying to |
| CONN_CONNECTED | Linea is connected |

#### 1.2.2.8 - (void) firmwareUpdateProgress: (int) *phase* percent:(int) *percent*

Notification sent when firmware update process advances.

Do not call any linea functions until firmware update is complete! During the firmware update notifications will be posted.

**Parameters**

| | |
|---|---|
| *phase* | update phase, one of: |

| | |
|---|---|
| UPDATE_INIT | Initializing firmware update |
| UPDATE_ERASE | Erasing flash memory |
| UPDATE_WRITE | Writing data |
| UPDATE_FINISH | Update complete |

| | |
|---|---|
| *percent* | firmware update progress in percents |

#### 1.2.2.9 - (void) magneticCardData: (NSString ∗) *track1* track2:(NSString ∗) *track2* track3:(NSString ∗) *track3*

Notification sent when magnetic card is successfuly read.

**Parameters**

| | |
|---|---|
| *track1* | - data contained in track 1 of the magnetic card or nil |
| *track2* | - data contained in track 2 of the magnetic card or nil |
| *track3* | - data contained in track 3 of the magnetic card or nil |

#### 1.2.2.10 - (void) magneticCardEncryptedData: (int) *encryption* data:(NSData ∗) *data*

Notification sent when magnetic card is successfuly read.

The data is being sent encrypted.

**Parameters**

| | |
|---|---|
| *encryption* | encryption algorithm used, one of: |

| | |
|---|---|
| 0 | AES 256 |
| 1 | IDTECH with DUKPT |

For AES256, after decryption, the result data will be as follows:

- Random data (4 bytes)
- Device identification text (16 ASCII characters, unused bytes are 0)
- Processed track data in the format: 0xF1 (track1 data), 0xF2 (track2 data) 0xF3 (track3 data). It is possible some of the tracks will be empty, then the identifier will not be present too, for example 0xF1 (track1 data) 0xF3 (track3 data)
- End of track data (byte 0x00)
- CRC16 (2 bytes) - the CRC is performed from the start of the encrypted block (the Random Data block) to the end of the track data (including the 0x00 byte). The data block is rounded to 16 bytes

In the more secure way, where the decryption key resides in a server only, the card read process will look something like:

- (User) swipes the card
- (iOS program) receives the data via magneticCardEncryptedData and sends to the server
- (iOS program)[optional] sends current Linea serial number along with the data received from magneticCard-EncryptedData. This can be used for data origin verification
- (Server) decrypts the data, extracts all the information from the fields
- (Server)[optional] if the ipod program have sent the Linea serial number before, the server compares the received serial number with the one that's inside the encrypted block
- (Server) checks if the card data is the correct one, i.e. all needed tracks are present, card is the same type as required, etc and sends back notification to the ipod program.

For IDTECH with DUKPT the data contains:

- DATA[0]: CARD TYPE: 0 - payment card
- DATA[1]: TRACK FLAGS
- DATA[2]: TRACK 1 LENGTH
- DATA[3]: TRACK 2 LENGTH
- DATA[4]: TRACK 3 LENGTH
- DATA[??]: TRACK 1 DATA MASKED
- DATA[??]: TRACK 2 DATA MASKED
- DATA[??]: TRACK 3 DATA
- DATA[??]: TRACK 1 AND TRACK 2 TDES ENCRYPTED
- DATA[??]: TRACK 1 SHA1 (0x14 BYTES)
- DATA[??]: TRACK 2 SHA1 (0x14 BYTES)
- DATA[??]: DUKPT SERIAL AND COUNTER (0x0A BYTES)

**Parameters**

| | |
|---|---|
| *data* | contains the encrypted card data |

#### 1.2.2.11 - (void) magneticCardEncryptedData: (int) *encryption* tracks:(int) *tracks* data:(NSData ∗) *data*

Notification sent when magnetic card is successfuly read.

The data is being sent encrypted.

**Parameters**

| | |
|---|---|
| *encryption* | encryption algorithm used, one of:<br>0 — AES 256<br>1 — IDTECH with DUKPT |

For AES256, after decryption, the result data will be as follows:

- Random data (4 bytes)
- Device identification text (16 ASCII characters, unused bytes are 0)
- Processed track data in the format: 0xF1 (track1 data), 0xF2 (track2 data) 0xF3 (track3 data). It is possible some of the tracks will be empty, then the identifier will not be present too, for example 0xF1 (track1 data) 0xF3 (track3 data)
- End of track data (byte 0x00)
- CRC16 (2 bytes) - the CRC is performed from the start of the encrypted block (the Random Data block) to the end of the track data (including the 0x00 byte). The data block is rounded to 16 bytes

In the more secure way, where the decryption key resides in a server only, the card read process will look something like:

- (User) swipes the card
- (iOS program) receives the data via magneticCardEncryptedData and sends to the server
- (iOS program)[optional] sends current Linea serial number along with the data received from magneticCard-EncryptedData. This can be used for data origin verification
- (Server) decrypts the data, extracts all the information from the fields
- (Server)[optional] if the ipod program have sent the Linea serial number before, the server compares the received serial number with the one that's inside the encrypted block
- (Server) checks if the card data is the correct one, i.e. all needed tracks are present, card is the same type as required, etc and sends back notification to the ipod program.

For IDTECH with DUKPT the data contains:

- DATA[0]: CARD TYPE: 0 - payment card
- DATA[1]: TRACK FLAGS
- DATA[2]: TRACK 1 LENGTH
- DATA[3]: TRACK 2 LENGTH
- DATA[4]: TRACK 3 LENGTH
- DATA[??]: TRACK 1 DATA MASKED

- DATA[??]: TRACK 2 DATA MASKED
- DATA[??]: TRACK 3 DATA
- DATA[??]: TRACK 1 AND TRACK 2 TDES ENCRYPTED
- DATA[??]: TRACK 1 SHA1 (0x14 BYTES)
- DATA[??]: TRACK 2 SHA1 (0x14 BYTES)
- DATA[??]: DUKPT SERIAL AND COUNTER (0x0A BYTES)

**Parameters**

| | |
|---|---|
| *tracks* | contain information which tracks are successfully read and inside the encrypted data as bit fields, bit 1 corresponds to track 1, etc, so value of 7 means all tracks are read |
| *data* | contains the encrypted card data |

#### 1.2.2.12 - (void) magneticCardEncryptedData: (int) *encryption* tracks:(int) *tracks* data:(NSData ∗) *data* track1masked:(NSString ∗) *track1masked* track2masked:(NSString ∗) *track2masked* track3:(NSString ∗) *track3*

Notification sent when magnetic card is successfuly read.

The data is being sent encrypted.

**Parameters**

| | |
|---|---|
| *encryption* | encryption algorithm used, one of:<br>0 — AES 256<br>1 — IDTECH with DUKPT |

For AES256, after decryption, the result data will be as follows:

- Random data (4 bytes)
- Device identification text (16 ASCII characters, unused bytes are 0)
- Processed track data in the format: 0xF1 (track1 data), 0xF2 (track2 data) 0xF3 (track3 data). It is possible some of the tracks will be empty, then the identifier will not be present too, for example 0xF1 (track1 data) 0xF3 (track3 data)
- End of track data (byte 0x00)
- CRC16 (2 bytes) - the CRC is performed from the start of the encrypted block (the Random Data block) to the end of the track data (including the 0x00 byte). The data block is rounded to 16 bytes

In the more secure way, where the decryption key resides in a server only, the card read process will look something like:

- (User) swipes the card
- (iOS program) receives the data via magneticCardEncryptedData and sends to the server
- (iOS program)[optional] sends current Linea serial number along with the data received from magneticCard-EncryptedData. This can be used for data origin verification
- (Server) decrypts the data, extracts all the information from the fields
- (Server)[optional] if the ipod program have sent the Linea serial number before, the server compares the received serial number with the one that's inside the encrypted block

- (Server) checks if the card data is the correct one, i.e. all needed tracks are present, card is the same type as required, etc and sends back notification to the ipod program.

For IDTECH with DUKPT the data contains:

- DATA[0]: CARD TYPE: 0 - payment card
- DATA[1]: TRACK FLAGS
- DATA[2]: TRACK 1 LENGTH
- DATA[3]: TRACK 2 LENGTH
- DATA[4]: TRACK 3 LENGTH
- DATA[??]: TRACK 1 DATA MASKED
- DATA[??]: TRACK 2 DATA MASKED
- DATA[??]: TRACK 3 DATA
- DATA[??]: TRACK 1 AND TRACK 2 TDES ENCRYPTED
- DATA[??]: TRACK 1 SHA1 (0x14 BYTES)
- DATA[??]: TRACK 2 SHA1 (0x14 BYTES)
- DATA[??]: DUKPT SERIAL AND COUNTER (0x0A BYTES)

**Parameters**

| | |
|---|---|
| *tracks* | contain information which tracks are successfully read and inside the encrypted data as bit fields, bit 1 corresponds to track 1, etc, so value of 7 means all tracks are read |
| *data* | contains the encrypted card data |
| *track1masked* | when possible, track1 data will be masked and returned here |
| *track2masked* | when possible, track2 data will be masked and returned here |

#### 1.2.2.13 - (void) magneticCardEncryptedRawData: (int) *encryption* data:(NSData ∗) *data*

Notification sent when magnetic card is successfuly read.

The raw card data is encrypted via the selected encryption algorithm. After decryption, the result data will be as follows:

- Random data (4 bytes)
- Device identification text (16 ASCII characters, unused bytes are 0)
- Track data: the maximum length of a single track is 704 bits (88 bytes), so track data contains 3x88 bytes
- CRC16 (2 bytes) - the CRC is performed from the start of the encrypted block (the Random Data block) to the end of the track data. The data block is rounded to 16 bytes

**Parameters**

| | |
|---|---|
| *encryption* | encryption algorithm used, one of:<br>0 — AES 256 |
| *data* | - Contains the encrypted raw card data |

#### 1.2.2.14 - (void) magneticCardRawData: (NSData ∗) *tracks*

Notification sent when magnetic card is successfuly read.

**Parameters**

| | |
|---|---|
| *tracks* | contains the raw magnetic card data. These are the bits directly from the magnetic head. The maximum length of a single track is 704 bits (88 bytes), so the command returns the 3 tracks as 3x88 bytes block |

#### 1.2.2.15 - (void) magneticJISCardData: (NSString ∗) *data*

Notification sent when JIS I & II magnetic card is successfuly read.

**Parameters**

| | |
|---|---|
| *data* | - data contained in the magnetic card |

#### 1.2.2.16 - (void) rfCardDetected: (int) *cardIndex* info:(DTRFCardInfo ∗) *info*

Notification sent when a new supported RFID card enters the field.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card, use this index with all subsequent commands to the card |
| *type* | card type, one of the CARD_∗ constants |
| *info* | information about the card |

#### 1.2.2.17 - (void) rfCardRemoved: (int) *cardIndex*

Notification sent when the card leaves the field.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card, use this index with all subsequent commands to the card |

## 1.3 General functions

Functions to connect/disconnect, set delegate, make sounds, update firmware, control various device settings.

### Functions

- (id) + Linea::sharedDevice

  *Creates and initializes new Linea class instance or returns already initalized one.*
- (void) - Linea::addDelegate:

  *Allows unlimited delegates to be added to a single class instance.*
- (void) - Linea::removeDelegate:

  *Removes delegate, previously added with addDelegate.*
- (void) - Linea::connect

  *Tries to connect to Linea in the background, connection status notifications will be passed through the delegate.*
- (void) - Linea::disconnect

  *Stops the sdk from trying to connect to Linea and breaks existing connection.*
- (BOOL) - **Linea::isPresent**
- (BOOL) - Linea::getBatteryCapacity:voltage:error:

  *Returns Linea's battery capacity.*
- (BOOL) - Linea::playSound:beepData:length:error:

  *Makes Linea plays a sound using the built-in speaker.*
- (BOOL) - Linea::getCharging:error:

  *Returns if Linea is charging the iOS device from it's own battery.*
- (BOOL) - Linea::setCharging:error:

  *Enables or disables Lines's capability to charge the handheld from it's own battery.*
- (NSDictionary ∗) - Linea::getFirmwareFileInformation:error:

  *Returns information about the specified firmware data.*
- (BOOL) - Linea::updateFirmwareData:error:

  *Updates Linea's firmware with specified firmware data.*
- (BOOL) - Linea::getSyncButtonMode:error:

  *Returns the current sync button mode.*
- (BOOL) - Linea::setSyncButtonMode:error:

  *Sets Linea's sync button mode.*

### 1.3.1 Detailed Description

Functions to connect/disconnect, set delegate, make sounds, update firmware, control various device settings.

### 1.3.2 Function Documentation

#### 1.3.2.1 - (void) addDelegate: (id) *newDelegate*

Allows unlimited delegates to be added to a single class instance.

This is useful in the case of global class and every view can use addDelegate when the view is shown and removeDelegate when no longer needs to monitor events

**Parameters**

| | |
|---|---|
| *newDelegate* | the delegate that will be notified of Linea events |

#### 1.3.2.2 - (void) connect

Tries to connect to Linea in the background, connection status notifications will be passed through the delegate.

Once connect is called, it will automatically try to reconnect until disconnect is called. Note that "connect" call works in background and will notify the caller of connection success via connectionState delegate. Do not assume the library has fully connected to the device after this call, but wait for the notification.

#### 1.3.2.3 - (BOOL) getBatteryCapacity: (int *) *capacity* voltage:(float *) *voltage* error:(NSError **) *error*

Returns Linea's battery capacity.

**Note**

Reading battery voltages during charging (both Linea charing and Linea charging the iPod) is unreliable!

**Parameters**

| | |
|---|---|
| *capacity* | returns battery capacity in percents, ranging from 0 when battery is dead to 100 when fully charged. Pass nil if you don't want that information |
| *voltage* | returns battery voltage in Volts, pass nil if you don't want that information |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.3.2.4 - (BOOL) getCharging: (BOOL *) *charging* error:(NSError **) *error*

Returns if Linea is charging the iOS device from it's own battery.

Linea firmware versions prior to 2.13 will return true if external charge is attached, 2.13 and later will return only if Linea's own battery is used for charging.

**Parameters**

| | |
|---|---|
| *charging* | returns TRUE if charging is enabled (from internal battery, external charging is omitted) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.3.2.5 - (NSDictionary *) getFirmwareFileInformation: (NSData *) *data* error:(NSError **) *error*

Returns information about the specified firmware data.

Based on it, and the connected Linea's name, model and firmware version you can chose to update or not the Linea's firmware

**Parameters**

| | |
|---|---|
| *data* | - firmware data |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

firmware information if function succeeded, nil otherwise. See Info∗ constants for possible keys in the returned dictionary.

#### 1.3.2.6 - (BOOL) getSyncButtonMode: (int ∗) *mode* error:(NSError ∗∗) *error*

Returns the current sync button mode.

See setSyncButtonMode for more detailed description. This setting is stored into flash memory and will persist.

**Note**

Although this function was made for Linea 1, that had hardware button to enter sync mode, it still works for enabling/disabling automated sync on Linea 4 and onward

**Parameters**

| | |
|---|---|
| *mode* | returns sync button mode, one of the:<br>BUTTON_DISABLED — Linea's will not perform synchronization when you press and hold the button for 3 seconds<br>BUTTON_ENABLED — Linea's will perform synchronization when you press and hold the button for 3 seconds |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.3.2.7 - (BOOL) playSound: (int) *volume* beepData:(int ∗) *data* length:(int) *length* error:(NSError ∗∗) *error*

Makes Linea plays a sound using the built-in speaker.

**Note**

A sample beep containing of 2 tones, each with 400ms duration, first one 2000Hz and second - 5000Hz will look int beepData[]={2000,400,5000,400}

**Parameters**

| | |
|---|---|
| *volume* | controls the volume (0-100). Currently have no effect |
| *data* | an array of integer values specifying pairs of tone(Hz) and duration(ms). |
| *length* | length in bytes of beepData array |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.3.2.8 - (void) removeDelegate: (id) *newDelegate*

Removes delegate, previously added with addDelegate.

**Parameters**

| | |
|---|---|
| *newDelegate* | the delegate that will be no longer be notified of Linea events |

#### 1.3.2.9 - (BOOL) setCharging: (BOOL) *enabled* error:(NSError **) *error*

Enables or disables Lines's capability to charge the handheld from it's own battery.

Charging can stop if Linea's battery goes too low.

While Linea can act as external battery for the iPod/iPhone, there are certain limitations if you decide to implement it. The internal battery is not big enough, so if the iPod/iPhone consumes a lot of power from it, it will go down very fast and force the firmware to cut the charge to prevent going down to dangerous levels. The proper use of this charging function depends on how the program, running on the iPod/iPhone, is used and how the iPod/iPhone is discharged

There are two possible ways to use Linea's charge:

- Emergency mode - in the case iPod/iPhone usage is designed in a way it will last long enough between charging sessions and using Linea's charge is not generally needed, the charge can be used if the iPod/iPhone for some reason goes too low (like <50%), so it is given some power to continue working until next charging. An example will be store, where devices are being charged every night, but extreme usage on some iPod drains the battery before the end of the shift. This is the less efficient way to charge it, also, Linea will refuse to start the charge if it's own battery goes below 3.8v, so depending on the usage, barcode type and if the barcode engine is set to work all the time, it may not be possible to start the charge.

- Max life mode - it is the case where both devices are required to operate as long as possible. Usually, the iPod/iPhone's battery will be drained way faster than Linea's, especially with wifi enabled programs and to keep both devices operating as long as possible, the charging should be desinged in a way so iPod/iPhone is able to use most of Linea's battery. This is possible, if you start charging when iPod/iPhone is almost full - at around 75-80% or higher. This way the iPod will consume small amount of energy, allowing our battery to slowly be used almost fully to charge it.

LineaDemo application contains sample implementation of max life mode charging.

**Note**

Reading battery voltages during charging is unreliable!
Enabling charge can fail if Linea's battery is low. Disabling charge will fail if there is external charger or usb cable attached.

**Parameters**

| | |
|---|---|
| *enabled* | TRUE to enable charging, FALSE to disable/stop it |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.3.2.10 - (BOOL) setSyncButtonMode: (int) *mode* error:(NSError **) *error*

Sets Linea's sync button mode.

This setting is stored into flash memory and will persist.

**Note**

Although this function was made for Linea 1, that had hardware button to enter sync mode, it still works for enabling/disabling automated sync on Linea 4 and onward

**Parameters**

| | |
|---|---|
| *mode* | button mode, one of the:<br>BUTTON_DISABLED — Linea's will not perform synchronization when you press and hold the button for 3 seconds<br>BUTTON_ENABLED (default) — Linea's will perform synchronization when you press and hold the button for 3 seconds |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.3.2.11 + (id) sharedDevice

Creates and initializes new Linea class instance or returns already initalized one.

Use this function, if you want to access the class from different places

**Returns**

shared class instance

#### 1.3.2.12 - (BOOL) updateFirmwareData: (NSData *) *data* error:(NSError **) *error*

Updates Linea's firmware with specified firmware data.

The firmware can only be upgraded or downgraded, if you send the same firmware version, then no update process will be started.

**Note**

Make sure the user does not interrupt the process or the device will be rendered unusable and can only be recovered via the special firmware update cable

**Parameters**

| | |
|---|---|
| *data* | the firmware data |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

## 1.4 Magnetic Stripe Reader Functions

Functions to work with the Linea's built-in magenetic card reader.

### Functions

- (BOOL) - Linea::msEnable:

  *Enables reading of magnetic cards.*
- (BOOL) - Linea::msDisable:

  *Disables magnetic card scanning started with msEnable.*
- (NSDictionary ∗) - Linea::msProcessFinancialCard:track2:

  *Helper function to parse financial card and extract the data - name, number, expiration date.*
- (BOOL) - Linea::msGetCardDataMode:error:

  *Returns the current magnetic card data mode.*
- (BOOL) - Linea::msSetCardDataMode:error:

  *Sets Linea's magnetic card data mode.*

### 1.4.1 Detailed Description

Functions to work with the Linea's built-in magenetic card reader.

### 1.4.2 Function Documentation

#### 1.4.2.1 - (BOOL) msDisable: (NSError ∗∗) *error*

Disables magnetic card scanning started with msEnable.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.4.2.2 - (BOOL) msEnable: (NSError ∗∗) *error*

Enables reading of magnetic cards.

Whenever a card is successfully read, the magneticCardData delegate will be called. Current magnetic card heads used in Linea consume so little power, that there is no drawback in leaving scanning enabled all the time.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.4.2.3 - (BOOL) msGetCardDataMode: (int *) *mode* error:(NSError **) *error*

Returns the current magnetic card data mode.

This setting is not persistent and is best to configure it upon connect.

**Returns**

card data mode, one of the:

| | |
|---|---|
| MS_PROCESSED_CARD_DATA | Card data will be processed and will be returned via call to magneticCardData |
| MS_RAW_CARD_DATA | Card data will not be processed and will be returned via call to magneticCardRawData |

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.4.2.4 - (NSDictionary *) msProcessFinancialCard: (NSString *) *track1* track2:(NSString *) *track2*

Helper function to parse financial card and extract the data - name, number, expiration date.

The function will extract as much information as possible.

**Parameters**

| | |
|---|---|
| *track1* | - track1 information or nil |
| *track2* | - track2 information or nil |

**Returns**

dictionary containing extracted data or nil if the data is invalid. Keys contained are:

| | |
|---|---|
| "accountNumber" | Account number |
| "cardholderName" | Cardholder name, as stored in the card |
| "expirationYear" | Expiration date - year |
| "expirationMonth" | Expiration date - month |
| "serviceCode" | Service code (if any) |
| "discretionaryData" | Discretionary data (if any) |
| "firstName" | Extracted cardholder's first name |
| "lastName" | Extracted cardholder's last name |

**Exceptions**

| | |
|---|---|
| *NSPortTimeoutException* | if there is no connection to Linea |

#### 1.4.2.5 - (BOOL) msSetCardDataMode: (int) *mode* error:(NSError **) *error*

Sets Linea's magnetic card data mode.

This setting is not persistent and is best to configure it upon connect.

**Parameters**

| | |
|---|---|
| *mode* | magnetic card data mode:<br>MS_PROCESSED_CARD_DATA — Card data will be processed and will be returned via call to magneticCardData<br>MS_RAW_CARD_DATA — Card data will not be processed and will be returned via call to magneticCardRawData |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

## 1.5 Barcode Reader Functions

Functions for scanning barcodes, various barcode settings and direct control of the barcode engine.

### Functions

- (NSString *) - Linea::barcodeType2Text:

  *Helper function to return string name of barcode type.*
- (BOOL) - Linea::barcodeStartScan:

  *Starts barcode engine.*
- (BOOL) - Linea::barcodeStopScan:

  *Stops ongoing scan started with barcodeStartScan.*
- (BOOL) - Linea::barcodeGetScanTimeout:error:

  *Returns the current scan timeout.*
- (BOOL) - Linea::barcodeSetScanTimeout:error:

  *Sets the scan timeout.*
- (BOOL) - Linea::barcodeGetScanButtonMode:error:

  *Returns the current scan button mode.*
- (BOOL) - Linea::barcodeSetScanButtonMode:error:

  *Sets Linea's scan button mode.*
- (BOOL) - Linea::barcodeSetScanBeep:volume:beepData:length:error:

  *Sets Linea's beep, which is used upon successful barcode scan.*
- (BOOL) - Linea::barcodeGetScanMode:error:

  *Returns the current scan mode.*
- (BOOL) - Linea::barcodeSetScanMode:error:

  *Sets Linea's barcode engine scan mode.*
- (BOOL) - Linea::barcodeEnableBarcode:enabled:error:

  *Enables or disables reading of specific barcode type.*
- (BOOL) - Linea::barcodeIsBarcodeEnabled:

  *Returns if the the engine is set to read the barcode type or not.*
- (BOOL) - Linea::barcodeIsBarcodeSupported:

  *Returns if the the engine can read the barcode type or not.*
- (BOOL) - Linea::barcodeGetTypeMode:error:

  *Returns the current barcode type mode.*
- (BOOL) - Linea::barcodeSetTypeMode:error:

  *Sets barcode type mode.*
- (BOOL) - Linea::barcodeEnginePowerControl:error:

  *Allows basic control over the power to the barcode engine.*
- (BOOL) - Linea::barcodeEnginePowerControl:maxTimeMinutes:error:

  *Allows basic control over the power to the barcode engine.*
- (BOOL) - Linea::barcodeEngineResetToDefaults:

  *Performs factory reset of the barcode module.*
- (BOOL) - Linea::barcodeOpticonSetInitString:error:

  *Allows for a custom initialization string to be sent to the Opticon barcode engine.*
- (BOOL) - Linea::barcodeOpticonSetParams:saveToFlash:error:

  *Sends configuration parameters directly to the opticon barcode engine.*
- (NSString *) - Linea::barcodeOpticonGetIdent:

  *Reads barcode engine's identification.*
- (BOOL) - Linea::barcodeOpticonUpdateFirmware:bootLoader:error:

  *Performs firmware update on the optiocon 2D barcode engines.*

- (BOOL) - Linea::barcodeCodeSetParam:value:error:

  *Sends configuration parameters directly to the code barcode engine.*
- (BOOL) - Linea::barcodeCodeGetParam:value:error:

  *Reads configuration parameters directly from the code barcode engine.*
- (BOOL) - Linea::barcodeCodeUpdateFirmware:data:error:

  *Performs firmware update on the Code 2D barcode engines.*
- (NSDictionary ∗) - **Linea::barcodeCodeGetInformation:**
- (BOOL) - Linea::barcodeIntermecSetInitData:error:

  *Allows for a custom initialization string to be sent to the Intermec barcode engine.*

### 1.5.1 Detailed Description

Functions for scanning barcodes, various barcode settings and direct control of the barcode engine.

### 1.5.2 Function Documentation

#### 1.5.2.1 - (BOOL) barcodeCodeGetParam: (int) *setting* value:(uint64_t ∗) *value* error:(NSError ∗∗) *error*

Reads configuration parameters directly from the code barcode engine.

Refer to the barcode engine documentation for supported parameters.

**Parameters**

| | |
|---|---|
| *setting* | the setting number |
| *value* | unpon success, the parameter value will be stored here |

**Returns**

TRUE if operation was successful

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.2 - (BOOL) barcodeCodeSetParam: (int) *setting* value:(uint64_t) *value* error:(NSError ∗∗) *error*

Sends configuration parameters directly to the code barcode engine.

Use this function with EXTREME care, you can easily render your barcode engine useless. Refer to the barcode engine documentation for supported parameters.

**Parameters**

| | |
|---|---|
| *setting* | the setting number |
| *value* | the value to write to |

**Returns**

TRUE if operation was successful

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.3 - (BOOL) barcodeCodeUpdateFirmware: (NSString ∗) *name* data:(NSData ∗) *data* error:(NSError ∗∗) *error*

Performs firmware update on the Code 2D barcode engines.

Barcode update can take very long time, it is best to call this function from a thread and update the user interface when firmwareUpdateProgress delegate is called

**Parameters**

| | |
|---|---|
| *name* | the exact name of the firmware file |
| *firmwareData* | firmware file data to load |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.4 - (BOOL) barcodeEnableBarcode: (int) *barcodeType* enabled:(BOOL) *enabled* error:(NSError ∗∗) *error*

Enables or disables reading of specific barcode type.

This setting is stored into the flash memory and will persists.

**Parameters**

| | |
|---|---|
| *barcodeType* | barcode type, one of the BAR_∗ constants with the exception of BAR_LAST. You can use BAR_ALL to enable or disable all barcode types at once |
| *enabled* | enables or disables reading of that barcode type |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.5 - (BOOL) barcodeEnginePowerControl: (BOOL) *engineOn* error:(NSError ∗∗) *error*

Allows basic control over the power to the barcode engine.

By default Linea manages barcode engine by turning it on when scan operation is needed, then turning off after 5 seconds of inactivity. There are situations, where barcode engine should stay on to give better user experience,

namely when using 2D barcode engine, which takes 1.7 seconds to start. This function is ignored for 1D barcode engines.

Be cautious using this function, if you pass TRUE to engineOn, the barcode engine will not turn off unless Linea is disconnected, program closes connection or iPod/iPhone goes to sleep, so it can drain the battery.

This setting does not persist, it is valid for current session only.

**Parameters**

| | |
|---|---|
| *engineOn* | TRUE will keep the engine powered on until the function is called with FALSE. In case of FALSE, Linea will work the usual way - powers on the engine just before scan operation. |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.6 - (BOOL) barcodeEnginePowerControl: (BOOL) *engineOn* maxTimeMinutes:(int) *maxTimeMinutes* error:(NSError **) *error*

Allows basic control over the power to the barcode engine.

By default Linea manages barcode engine by turning it on when scan operation is needed, then turning off after 5 seconds of inactivity. There are situations, where barcode engine should stay on to give better user experience, namely when using 2D barcode engine, which takes 1.7 seconds to start. This function is ignored for 1D barcode engines.

Be cautious using this function, if you pass TRUE to engineOn, the barcode engine will not turn off unless Linea is disconnected, program closes connection or iPod/iPhone goes to sleep, so it can drain the battery.

This setting does not persist, it is valid for current session only.

**Parameters**

| | |
|---|---|
| *engineOn* | TRUE will keep the engine powered on until the function is called with FALSE. In case of FALSE, Linea will work the usual way - powers on the engine just before scan operation. |
| *maxTime-Minutes* | the maximum idle time the engine will be kept on, in minutes. After that time elapses, the engine will be turned off to conserve power. Recommended value - 60 min. Setting the time is supported only on version 2.64 and later, on older firmware versions the time parameter is ignored. |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.7 - (BOOL) barcodeEngineResetToDefaults: (NSError **) *error*

Performs factory reset of the barcode module.

This function is taxing, slow and should not be called often, emergency use only.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.8 - (BOOL) barcodeGetScanButtonMode: (int *) *mode* error:(NSError **) *error*

Returns the current scan button mode.

See barcodeSetScanButtonMode for more detailed description. This setting is not persistent and is best to configure it upon connect.

**Parameters**

| | |
|---|---|
| *mode* | returns scan button mode, one of the:<br>BUTTON_DISABLED — Linea's button will become inactive<br>BUTTON_ENABLED — Linea's button will triger barcode scan when pressed |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.9 - (BOOL) barcodeGetScanMode: (int *) *mode* error:(NSError **) *error*

Returns the current scan mode.

This setting is not persistent and is best to configure it upon connect.

**Parameters**

| | |
|---|---|
| *mode* | returns scanning mode, one of the MODE_* constants |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.10 - (BOOL) barcodeGetScanTimeout: (int *) *timeout* error:(NSError **) *error*

Returns the current scan timeout.

See barcodeSetScanTimeout for more detailed description. This setting is not persistent and is best to configure it upon connect.

**Parameters**

| | |
|---|---|
| *timeout* | returns scan timeout in seconds |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.11 - (BOOL) barcodeGetTypeMode: (int *) *mode* error:(NSError **) *error*

Returns the current barcode type mode.

See barcodeSetTypeMode for more detailed description. This setting will not persists.

**Parameters**

| | |
|---|---|
| *mode* | returns barcode type mode, one of the:<br>BARCODE_TYPE_DEFAULT — default barcode types, listed in BARCODES enumeration<br>BARCODE_TYPE_EXTENDED — extended barcode types, listed in BARCODES_EX enumeration |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.12 - (BOOL) barcodeIntermecSetInitData: (NSData *) *data* error:(NSError **) *error*

Allows for a custom initialization string to be sent to the Intermec barcode engine.

The data is sent directly, if the barcode is currently powered on, and every time it gets initialized. The setting does not persists, so it is best this command is called upon new connection with Linea.

**Parameters**

| | |
|---|---|
| *data* | barcode engine initialization data (consult barcode engine manual) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.13 - (BOOL) barcodeIsBarcodeEnabled: (int) *type*

Returns if the the engine is set to read the barcode type or not.

**Parameters**

| | |
|---|---|
| *type* | barcode type, one of the BAR_* constants with the exception of BAR_ALL and BAR_LAST |

**Returns**

TRUE if the barcode is enabled

#### 1.5.2.14 - (BOOL) barcodeIsBarcodeSupported: (int) *type*

Returns if the the engine can read the barcode type or not.

**Parameters**

| | |
|---|---|
| *type* | barcode type, one of the BAR_∗ constants with the exception of BAR_ALL and BAR_LAST |

**Returns**

TRUE if the barcode is supported

#### 1.5.2.15 - (NSString ∗) barcodeOpticonGetIdent: (NSError ∗∗) *error*

Reads barcode engine's identification.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

opticon engine ident string if function succeeded, nil otherwise

#### 1.5.2.16 - (BOOL) barcodeOpticonSetInitString: (NSString ∗) *data* error:(NSError ∗∗) *error*

Allows for a custom initialization string to be sent to the Opticon barcode engine.

The string is sent directly, if the barcode is currently powered on, and every time it gets initialized. The setting does not persists, so it is best this command is called upon new connection with Linea.

**Parameters**

| | |
|---|---|
| *data* | barcode engine initialization data (consult barcode engine manual) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.17 - (BOOL) barcodeOpticonSetParams: (NSString ∗) *data* saveToFlash:(BOOL) *saveToFlash* error:(NSError ∗∗) *error*

Sends configuration parameters directly to the opticon barcode engine.

Use this function with EXTREME care, you can easily render your barcode engine useless. Refer to the barcode engine documentation on supported commands.

The function encapsulates the data with the ESC and CR so you don't have to send them. It optionally sends Z2 after the command to ensure settings are stored in the flash.

You can send multiple parameters with a single call if you format them as follows:

- commands that take 2 symbols can be sent without any delimiters, like: "C1C2C3"
- commands that take 3 symbols should be prefixed by [, like: "C1[C2AC3" (in this case commands are C1, C2A and C3
- commands that take 4 symbols should be prefixed by ], like: "C1C2]C3AB" (in this case commands are C1, C2 and C3AB

**Parameters**

| | |
|---|---|
| *data* | command string |
| *saveToFlash* | if TRUE, command also saves the settings to flash. Saving setting is slower, so should be in ideal case executed only once and the program to remember it. The scanner's power usually gets cut when Linea goes to sleep - 5 seconds of idle time, so any non-stored to flash settings are lost, but if barcodeEnginePowerControl:TRUE is used on 2D engine, then even non-saved to flash settings will persist until device disconnects (iOS goes to sleep, physical disconnect) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.18 - (BOOL) barcodeOpticonUpdateFirmware: (NSData ∗) *firmwareData* bootLoader:(BOOL) *bootLoader* error:(NSError ∗∗) *error*

Performs firmware update on the optiocon 2D barcode engines.

Barcode update can take very long time, it is best to call this function from a thread and update the user interface when firmwareUpdateProgress delegate is called

**Parameters**

| | |
|---|---|
| *firmwareData* | firmware file data to load |
| *bootloader* | TRUE if you are going to update bootloader, FALSE if normal firmware |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.19 - (BOOL) barcodeSetScanBeep: (BOOL) *enabled* volume:(int) *volume* beepData:(int ∗) *data* length:(int) *length* error:(NSError ∗∗) *error*

Sets Linea's beep, which is used upon successful barcode scan.

This setting is not persistent and is best to configure it upon connect.

**Note**

A sample beep containing of 2 tones, each with 400ms duration, first one 2000Hz and second - 5000Hz will look int beepData[]={2000,400,5000,400}

**Parameters**

| | |
|---|---|
| *enabled* | turns on or off beeping |
| *volume* | controls the volume (0-100). Currently have no effect |
| *data* | an array of integer values specifying pairs of tone(Hz) and duration(ms). |
| *length* | length in bytes of beepData array |

**Exceptions**

| | |
|---|---|
| *NSPortTimeoutException* | if there is no connection to Linea |
| *NSInvalidArgument-Exception* | if some of the input parameters are wrong |

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.20 - (BOOL) barcodeSetScanButtonMode: (int) *mode* error:(NSError ∗∗) *error*

Sets Linea's scan button mode.

This setting is not persistent and is best to configure it upon connect.

**Parameters**

| | |
|---|---|
| *mode* | button mode, one of the:<br>BUTTON_DISABLED — Linea's button will become inactive<br>BUTTON_ENABLED — Linea's button will triger barcode scan when pressed |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.21 - (BOOL) barcodeSetScanMode: (int) *mode* error:(NSError ∗∗) *error*

Sets Linea's barcode engine scan mode.

This setting is not persistent and is best to configure it upon connect.

**Parameters**

| | |
|---|---|
| *mode* | scanning mode, one of the MODE_∗ constants |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.22 - (BOOL) barcodeSetScanTimeout: (int) *timeout* error:(NSError ∗∗) *error*

Sets the scan timeout.

This it the max time that the laser will be on in single scan mode, or the time without scanning that will force the laser off in multi scan mode. This setting is not persistent and is best to configure it upon connect.

**Parameters**

| | |
|---|---|
| *timeout* | barcode engine timeout in seconds [1-60] or 0 to disable timeout. Default is 0 |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.23 - (BOOL) barcodeSetTypeMode: (int) *mode* error:(NSError **) *error*

Sets barcode type mode.

Linea can return barcode type from the default list (listed in BARCODES) or extended one (listed in BARCODES_EX). The extended one is superset to the default, so current programs will be mostly unaffected if they switch from default to extended (with the exception of barcodes like UPC-A and UPC-E, which will be returned as UPC in the default list, but proper types in the extended. This setting will not persists.

**Parameters**

| | |
|---|---|
| *mode* | barcode type mode, one of the:<br>BARCODE_TYPE_DEFAULT (default) — default barcode types, listed in BARCODES enumeration<br>BARCODE_TYPE_EXTENDED — extended barcode types, listed in BARCODES_EX enumeration |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.24 - (BOOL) barcodeStartScan: (NSError **) *error*

Starts barcode engine.

In single scan mode the laser will be on until barcode is successfully read, the timeout elapses (set via call to barcodeSetScanTimeout) or if barcodeStopScan is called. In multi scan mode the laser will stay on even if barcode is successfully read allowing series of barcodes to be scanned within a single read session. The scanning will stop if no barcode is scanned in the timeout interval (set via call to barcodeSetScanTimeout) or if barcodeStopScan is called.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.25 - (BOOL) barcodeStopScan: (NSError **) *error*

Stops ongoing scan started with barcodeStartScan.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.5.2.26 - (NSString ∗) barcodeType2Text: (int) *barcodeType*

Helper function to return string name of barcode type.

**Parameters**

| | |
|---|---|
| *barcodeType* | barcode type returned from scanBarcode |

**Returns**

barcode type name

## 1.6 Cryptographic & Security Functions

Starting from firmware 2.13, Linea provides strong cryptographic support for magnetic card data.

### Functions

- (NSData *) - Linea::cryptoRawGenerateRandomData:

  *Generates 16 byte block of random numbers, required for some of the other crypto functions.*
- (BOOL) - Linea::cryptoRawSetKey:encryptedData:keyVersion:keyFlags:error:
- (BOOL) - Linea::cryptoSetKey:key:oldKey:keyVersion:keyFlags:error:

  *Used to store AES256 keys into Linea internal memory.*
- (BOOL) - Linea::cryptoGetKeyVersion:keyVersion:error:

  *Returns key version.*
- (NSData *) - Linea::cryptoRawAuthenticateDevice:error:
- (BOOL) - Linea::cryptoAuthenticateDevice:error:
- (BOOL) - Linea::cryptoRawAuthenticateHost:error:
- (BOOL) - Linea::cryptoAuthenticateHost:error:

### 1.6.1 Detailed Description

Starting from firmware 2.13, Linea provides strong cryptographic support for magnetic card data. The encryption is supported on all Linea devices, from software point of view they are all the same, but provide different levels of hardware/firmware security.

An overview of the security, provided by Linea (see each of the crypto functions for further detail):

Hardware/Firmware:

For magnetic card encryption Linea is using AES256, which is the current industry standard encryption algorithm. The encryption key resides in volatile, battery powered ram inside Linea's cpu (for Linea 1.5 onward) and is being lost if anyone tries to break in the Linea device in order to prevent the key from being stolen. Magnetic card data, along with device serial number and some random bytes (to ensure every packet will be different) are being sent to the iOS program in an encrypted way.

Software:

Currently there are 2 types of keys, that can be loaded into Linea:

- AUTHENTICATION KEY - used for device authentication (for example the program can lock itself to work with very specific Linea device) and encryption of the firmware
- ENCRYPTION KEY - used for magnetic card data encryption. To use msr encryption, you don't need to set the AUTHENTICATION KEY.

Keys: The keys can be set/changed in two ways:

1. Using plain key data - this method is easy to use, but less secure, as it relies on program running on iPod/iPhone to have the key inside, an attacker could compromise the system and extract the key from device's memory. Call cryptoSetKey to set the keys this way. If there is an existing key of the same type inside Linea, you have to pass it too.

2. Using encrypted key data - this method is harder to implement, but provides better security - the key data, encrypted with old key data is sent from a server in secure environment to the program, running on the iOS, then the program forwards it to the Linea. The program itself have no means to decrypt the data, so an attacker can't possibly extract the key. Refer to cryptoSetKey documentation for more detailed description of the loading process.

The initial loading of the keys should always be done in a secure environment.

Magnetic card encryption:

Once ENCRYPTION KEY is set, all magnetic card data gets encrypted, and is now sent via magneticCardEncryptedData instead. The LineaDemo program contains sample code to decrypt the data block and extract the contents - the serial number and track data.

As with keys, card data can be extracted on the iOS device itself (less secure, the application needs to have the key inside) or be sent to a secure server to be processed. Note, that the encrypted data contains Linea's serial number too, this can be used for Data Origin Verification, to be sure someone is not trying to mimic data, coming from another device.

Demo program: the sample program now have "Crypto" tab, where key management can be tested:

- New AES 256 key - type in the key you want to set (or change to)
- Old AES 256 key - type in the previous key, or leave blank if you set the key for the first time

[SET AUTHENTICATION KEY] and [SET ENCRYPTION KEY] buttons allow you to use the key info in the text fields above to set the key.

- Decryption key - type in the key, which the demo program will use to try to decrypt card data. This field should contain the ENCRYPTION KEY, or something random, if you want to test failed card data decryption.

### 1.6.2 Function Documentation

#### 1.6.2.1 - (BOOL) cryptoAuthenticateDevice: (NSData *) *key* error:(NSError **) *error*

**Note**

Check out the cryptoRawAuthenticateDevice function, if you want to not use the key inside the mobile device.

Generates random data, uses the key to encrypt it, then encrypts the same data with the stored authentication key inside Linea and returns true if both data matches.

The idea: if a program wants to work with specific Linea device, it sets AES256 authentication key once, then on every connect the program uses cryptoAuthenticateDevice with that key. If Linea contains no key, or the key is different, the function will return FALSE. This does not block Linea from operation, what action will be taken if devices mismatch depends on the program.

**Parameters**

| | |
|---|---|
| *key* | 32 bytes AES256 key |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if if Linea contains the same authentication key, FALSE otherwise

#### 1.6.2.2 - (BOOL) cryptoAuthenticateHost: (NSData *) *key* error:(NSError **) *error*

**Note**

Check out the cryptoRawAuthenticateHost function, if you want to not use the key inside the mobile device.

Generates random data, uses the key to encrypt it, then sends to Linea to verify against it's internal authentication key. If both keys match, return value is TRUE. This function is used so that Linea knows a "real" device is currently connected, before allowing some functionality. Currently firmware update is protected by this function, once authentication key is set, you have to use it or cryptoRawAuthenticateHost before you attempt firmware update, or it will error out.

**Parameters**

| | |
|---|---|
| *key* | 32 bytes AES256 key |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if Linea contains the same authentication key, FALSE otherwise

#### 1.6.2.3 - (BOOL) cryptoGetKeyVersion: (int) *keyID* keyVersion:(uint32_t *) *keyVersion* error:(NSError **) *error*

Returns key version.

Valid key ID:

- KEY_AUTHENTICATION - if set, you can use authentication functions - cryptoRawAuthenticateDevice or cryptoAuthenticateDevice. Firmware updates will require authentication too
- KEY_ENCRYPTION - if set, magnetic card data will come encrypted via magneticCardEncryptedData or magneticCardEncryptedRawData

**Parameters**

| | |
|---|---|
| *keyVersion* | returns key version or 0 if the key is not present (key versions are available in firmware 2.43 or later) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.6.2.4 - (NSData *) cryptoRawAuthenticateDevice: (NSData *) *randomData* error:(NSError **) *error*

**Note**

RAW crypto functions are harder to use and require more code, but are created to allow no secret keys to reside on the device, but all the operations can be executed with data, sent from a secure server. See cryptoAuthenticateDevice if you plan to use the key in the mobile device.

Encrypts a 16 bytes block of random data with the stored authentication key and returns the result.

The idea: if a program wants to work with specific Linea device, it sets AES256 authentication key once, then on every connect the program generates random 16 byte block of data, encrypts it internally with the said key, then encrypts it with linea too and compares the result. If that Linea contains no key, or the key is different, the resulting data will totally differ from the one generated. This does not block Linea from operation, what action will be taken if devices mismatch depends on the program.

**Parameters**

| | |
|---|---|
| *randomData* | 16 bytes block of data (presumably random bytes) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

random data, encrypted with the Linea authentication key if function succeeded, nil otherwise

#### 1.6.2.5 - (BOOL) cryptoRawAuthenticateHost: (NSData *) *encryptedRandomData* error:(NSError **) *error*

**Note**

RAW crypto functions are harder to use and require more code, but are created to allow no secret keys to reside on the device, but all the operations can be executed with data, sent from a secure server. See cryptoAuthenticateHost if you plan to use the key in the mobile device.

Tries to decrypt random data, generated from cryptoRawGenerateRandomData with the stored internal authentication key and returns the result. This function is used so that Linea knows a "real" device is currently connected, before allowing some functionality. Currently firmware update is protected by this function, once authentication key is set, you have to use it or cryptoAuthenticateHost before you attempt firmware update, or it will error out.

The idea (considering the iOS device does not have the keys inside, but depends on server):

- (iOS program) generates random data using cryptoRawGenerateRandomData and sends to the server
- (Server) encrypts the random data with the same AES256 key that is in the Linea and sends back to the iOS program
- (iOS program) uses cryptoRawAuthenticateHost to authenticate with the data, an exception will be generated if authentication fails.

**Parameters**

| | |
|---|---|
| *encrypted-RandomData* | 16 bytes block of encrypted data |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.6.2.6 - (NSData *) cryptoRawGenerateRandomData: (NSError **) *error*

Generates 16 byte block of random numbers, required for some of the other crypto functions.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

16 bytes of random numbers if function succeeded, nil otherwise

#### 1.6.2.7 - (BOOL) cryptoRawSetKey: (int) *keyID* encryptedData:(NSData *) *encryptedData* keyVersion:(uint32_t) *keyVersion* keyFlags:(uint32_t) *keyFlags* error:(NSError **) *error*

**Note**

RAW crypto functions are harder to use and require more code, but are created to allow no secret keys to reside on the device, but all the operations can be executed with data, sent from a secure server. See cryptoSetKey if you plan to use the key in the mobile device.

Used to store AES256 keys into Linea internal memory. Valid keys that can be set:

- KEY_AUTHENTICATION - if set, you can use authentication functions - cryptoRawAuthenticateDevice or cryptoAuthenticateDevice. Firmware updates will require authentication too

- KEY_ENCRYPTION - if set, magnetic card data will come encrypted via magneticCardEncryptedData or magneticCardEncryptedRawData

Generally the key loading process, using "Raw" commands, a program on the iOS device and a server which holds the keys will look similar to:

- (iOS program) calls cryptoRawGenerateRandomData to get 16 bytes block of random data and send these to the server
- (Server) creates byte array of 48 bytes consisting of: [RANDOM DATA: 16 bytes][KEY DATA: 32 bytes]
- (Server) if there is current encryption key set on the Linea (if you want to change existing key) the server encrypts the 48 bytes block with the OLD key
- (Server) sends the result data back to the program
- (iOS program) calls cryptoRawSetKey with KEY_ENCRYPTION and the data it received from the server
- (Linea) tries to decrypt the key data if there was already key present, then extracts the key, verifies the random data and if everything is okay, sets the key

**Parameters**

| | |
|---|---|
| *keyID* | the key type to set - KEY_AUTHENTICATION or KEY_ENCRYPTION |
| *encryptedData* | - 48 bytes that consists of 16 bytes random numbers received via call to cryptoRawGenerate-RandomData and 32 byte AES256 key. If there has been previous key of the same type, then all 48 bytes should be encrypted with it. |
| *keyVersion* | - the version of the key. On firmware versions less than 2.43 this parameter is ignored and key version is considered to be 0x00000000. Key version is useful for the program to determine what key is inside the head. |
| *keyFlags* | - optional key flags, supported on ver 2.58 and onward |

- KEY_AUTHENTICATION:

| | |
|---|---|
| BIT 1 | If set to 1, scanning barcodes, reading magnetic card and using the bluetooth module are locked and have to be unlocked with cryptoAuthenticate-Host/cryptoRawAuthenticateHost upon every reinsert of the device |

- KEY_ENCRYPTION: No flags are supported

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.6.2.8 - (BOOL) cryptoSetKey: (int) *keyID* key:(NSData ∗) *key* oldKey:(NSData ∗) *oldKey* keyVersion:(uint32_t) *keyVersion* keyFlags:(uint32_t) *keyFlags* error:(NSError ∗∗) *error*

Used to store AES256 keys into Linea internal memory.

Valid keys that can be set:

- KEY_AUTHENTICATION - if set, you can use authentication functions - cryptoRawAuthenticateDevice or cryptoAuthenticateDevice. Firmware updates will require authentication too
- KEY_ENCRYPTION - if set, magnetic card data will come encrypted via magneticCardEncryptedData or magneticCardEncryptedRawData

**Parameters**

| | |
|---|---|
| *keyID* | the key type to set - KEY_AUTHENTICATION or KEY_ENCRYPTION |
| *key* | 32 bytes AES256 key to set |
| *oldKey* | 32 bytes AES256 key that was previously used, or null if there was no previous key. The old key should match the new key, i.e. if you are setting KEY_ENCRYPTION, then you should pass the old KEY_ENCRYPTION. |
| *keyVersion* | - the version of the key. On firmware versions less than 2.43 this parameter is ignored and key version is considered to be 0x00000000. Key version is useful for the program to determine what key is inside the head. |
| *keyFlags* | - optional key flags, supported on ver 2.58 and onward |

- KEY_AUTHENTICATION:

| | |
|---|---|
| BIT 1 | If set to 1, scanning barcodes, reading magnetic card and using the bluetooth module are locked and have to be unlocked with cryptoAuthenticate-Host/cryptoRawAuthenticateHost upon every reinsert of the device |

- KEY_ENCRYPTION: No flags are supported

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

## 1.7 Bluetooth Functions

Functions to work with Linea's built-in bluetooth module.

### Functions

- (BOOL) - Linea::btGetEnabled:error:

  *Returns bluethooth module status.*
- (BOOL) - Linea::btSetEnabled:error:

  *Enables or disables bluetooth module.*
- (BOOL) - Linea::btWrite:length:error:

  *Sends data to the connected remote device.*
- (BOOL) - Linea::btWrite:error:

  *Sends data to the connected remote device.*
- (int) - Linea::btRead:length:timeout:error:

  *Tries to read data from the connected remote device for specified timeout.*
- (NSString ∗) - Linea::btReadLine:error:

  *Tries to read string data, ending with CR/LF up to specifed timeout.*
- (NSString ∗) - Linea::btGetLocalName:

  *Retrieves local bluetooth name, this is the name that Linea will report to bluetooth discovery requests.*
- (NSArray ∗) - Linea::btDiscoverDevices:maxTime:codTypes:error:

  *Performs synchronous discovery of the nearby bluetooth devices.*
- (NSString ∗) - Linea::btGetDeviceName:error:

  *Queries device name given the address, this function complements the btDiscoverDevices/btDiscoverPrinters and as such is not recommended, use btDiscoverDevicesInBackground instead.*
- (BOOL) - Linea::btDiscoverDevicesInBackground:maxTime:codTypes:error:

  *Performs background discovery of nearby bluetooth devices.*
- (BOOL) - Linea::btDiscoverSupportedDevicesInBackground:maxTime:filter:error:

  *Performs background discovery of nearby supported bluetooth devices.*
- (NSArray ∗) - Linea::btDiscoverPrinters:maxTime:error:

  *Performs discovery of supported printers.*
- (BOOL) - Linea::btDiscoverPrintersInBackground:maxTime:error:

  *Performs background discovery of supported printers.*
- (BOOL) - Linea::btDiscoverPrintersInBackground:

  *Performs background discovery of supported printers.*
- (NSArray ∗) - Linea::btDiscoverPinpads:maxTime:error:

  *Performs discovery of supported pinpads.*
- (BOOL) - Linea::btDiscoverPinpadsInBackground:maxTime:error:

  *Performs background discovery of supported printers.*
- (BOOL) - Linea::btDiscoverPinpadsInBackground:

  *Performs background discovery of supported printers.*
- (BOOL) - Linea::btConnect:pin:error:

  *Tries to connect to remote device.*
- (BOOL) - Linea::btDisconnect:error:

  *Disconnects from remote device.*
- (BOOL) - Linea::btEnableWriteCaching:error:

  *Enables or disables write caching on the bluetooth stream.*
- (BOOL) - Linea::btSetDataNotificationMaxTime:maxLength:sequenceData:error:

  *Sets the conditions to fire the NSStreamEventHasBytesAvailable event on bluetooth streams.*

### Properties

- NSInputStream * Linea::btInputStream

  *Bluetooth input stream, you can use it after connecting with btConnect.*
- NSOutputStream * Linea::btOutputStream

  *Bluetooth output stream, you can use it after connecting with btConnect.*

## 1.7.1 Detailed Description

Functions to work with Linea's built-in bluetooth module.

## 1.7.2 Function Documentation

### 1.7.2.1 - (BOOL) btConnect: (NSString *) *address* pin:(NSString *) *pin* error:(NSError **) *error*

Tries to connect to remote device.

Once connection is established, use bluetooth streams to read/write to the remote device.

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *address* | bluetooth address returned from btDiscoverDevices/btDiscoverPrinters |
| *pin* | PIN code if needed, or nil to try unencrypted connection |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

### 1.7.2.2 - (BOOL) btDisconnect: (NSString *) *address* error:(NSError **) *error*

Disconnects from remote device.

Currently, due to bluetooth module limitation disconnect actually performs module power off and on, so the remote device may still hold on connected state for a while

**Parameters**

| | |
|---|---|
| *address* | bluetooth address returned from btDiscoverDevices/btDiscoverPrinters |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

### 1.7.2.3 - (NSArray *) btDiscoverDevices: (int) *maxDevices* maxTime:(double) *maxTime* codTypes:(int) *codTypes* error:(NSError **) *error*

Performs synchronous discovery of the nearby bluetooth devices.

This function is not recommended to be called on the main thread, use btDiscoverDevicesInBackground instead.

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *maxDevices* | the maximum results to return |
| *maxTime* | the max time to discover, in seconds. Actual time may vary. |
| *codTypes* | bluetooth Class Of Device to look for or 0 to search for all bluetooth devices |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

array of strings of bluetooth addresses if function succeeded, nil otherwise

#### 1.7.2.4 - (BOOL) btDiscoverDevicesInBackground: (int) *maxDevices* maxTime:(double) *maxTime* codTypes:(int) *codTypes* error:(NSError ∗∗) *error*

Performs background discovery of nearby bluetooth devices.

The discovery status and devices found will be sent via delegate notifications

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *maxDevices* | the maximum results to return |
| *maxTime* | the max time to discover, in seconds. Actual time may vary. |
| *codTypes* | bluetooth Class Of Device to look for or 0 to search for all bluetooth devices |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.7.2.5 - (NSArray ∗) btDiscoverPinpads: (int) *maxDevices* maxTime:(double) *maxTime* error:(NSError ∗∗) *error*

Performs discovery of supported pinpads.

These include MPED-400 and PPAD1.

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *maxDevices* | the maximum results to return |
| *maxTime* | the max time to discover, in seconds. Actual time may vary. |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

array of strings containing bluetooth device address and bluetooth device name, i.e. if 2 devices are found, the list will contain "address 1",@"name 1",@"address 2",@"name 2" if function succeeded, nil otherwise

#### 1.7.2.6 - (BOOL) btDiscoverPinpadsInBackground: (NSError ∗∗) *error*

Performs background discovery of supported printers.

These include MPED-400 and PPAD1. The discovery status and devices found will be sent via delegate notifications

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.7.2.7 - (BOOL) btDiscoverPinpadsInBackground: (int) *maxDevices* maxTime:(double) *maxTime* error:(NSError ∗∗) *error*

Performs background discovery of supported printers.

These include MPED-400 and PPAD1. The discovery status and devices found will be sent via delegate notifications

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *maxDevices* | the maximum results to return, default is 4 |
| *maxTime* | the max time to discover, in seconds. Actual time may vary. |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.7.2.8 - (NSArray ∗) btDiscoverPrinters: (int) *maxDevices* maxTime:(double) *maxTime* error:(NSError ∗∗) *error*

Performs discovery of supported printers.

These include PP-60, DPP-250, DPP-350, SM-112, DPP-450.

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *maxDevices* | the maximum results to return |
| *maxTime* | the max time to discover, in seconds. Actual time may vary. |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

array of strings containing bluetooth device address and bluetooth device name, i.e. if 2 devices are found, the list will contain "address 1",@"name 1",@"address 2",@"name 2" if function succeeded, nil otherwise

#### 1.7.2.9 - (BOOL) btDiscoverPrintersInBackground: (NSError ∗∗) *error*

Performs background discovery of supported printers.

These include PP-60, DPP-250, DPP-350, SM-112, DPP-450. The discovery status and devices found will be sent via delegate notifications

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.7.2.10 - (BOOL) btDiscoverPrintersInBackground: (int) *maxDevices* maxTime:(double) *maxTime* error:(NSError ∗∗) *error*

Performs background discovery of supported printers.

These include PP-60, DPP-250, DPP-350, SM-112, DPP-450. The discovery status and devices found will be sent via delegate notifications

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *maxDevices* | the maximum results to return, default is 4 |
| *maxTime* | the max time to discover, in seconds. Actual time may vary. |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.7.2.11 - (BOOL) btDiscoverSupportedDevicesInBackground: (int) *maxDevices* maxTime:(double) *maxTime* filter:(int) *filter* error:(NSError ∗∗) *error*

Performs background discovery of nearby supported bluetooth devices.

Supported devices are the ones some of the sdks has built-in support for - printers and pinpads. The discovery status and devices found will be sent via delegate notifications

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *maxDevices* | the maximum results to return |
| *maxTime* | the max time to discover, in seconds. Actual time may vary. |
| *filter* | filter of which devices to discover, a combination of one or more of BLUETOOT_FILTER_∗ constants or BLUETOOTH_FILTER_ALL to get all supported devices |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.7.2.12 - (BOOL) btEnableWriteCaching: (BOOL) *enabled* error:(NSError ∗∗) *error*

Enables or disables write caching on the bluetooth stream.

When enabled the writes gets cached and send on bigger chunks, reducing substantially the time taken, if you are sending lot of data in small parts. Write caching has negative effect on the speed if your bluetooth communication is based on request/response format or packets, in this case every write operation will get delayed, resulting in very poor throughput.

**Parameters**

| | |
|---|---|
| *enable* | enable or disable write caching, by default it is disabled |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.7.2.13 - (NSString ∗) btGetDeviceName: (NSString ∗) *address* error:(NSError ∗∗) *error*

Queries device name given the address, this function complements the btDiscoverDevices/btDiscoverPrinters and as such is not recommended, use btDiscoverDevicesInBackground instead.

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *address* | bluetooth address returned from btDiscoverDevices/btDiscoverPrinters |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

bluetooth device name if function succeeded, nil otherwise

#### 1.7.2.14 - (BOOL) btGetEnabled: (BOOL *) *enabled* error:(NSError **) *error*

Returns bluethooth module status.

**Note**

When bluetooth module is enabled, access to the barcode engine is not possible!

**Parameters**

| | |
|---|---|
| *enabled* | returns TRUE if bluetooth module is enabled, FALSE otherwise |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.7.2.15 - (NSString *) btGetLocalName: (NSError **) *error*

Retrieves local bluetooth name, this is the name that Linea will report to bluetooth discovery requests.

**Note**

this function cannot be called once connection to remote device was established

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

bluetooth name if function succeeded, nil otherwise

#### 1.7.2.16 - (int) btRead: (void *) *data* length:(int) *length* timeout:(double) *timeout* error:(NSError **) *error*

Tries to read data from the connected remote device for specified timeout.

**Note**

You can use bluethooth streams instead

**Parameters**

| | |
|---|---|
| *data* | data buffer, where the result will be stored |
| *length* | maximum amount of bytes to wait for |
| *timeout* | maximim timeout in seconds to wait for data |

**Returns**

the

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

actual number of bytes stored in the data buffer if function succeeded, -1 otherwise

#### 1.7.2.17 - (NSString ∗) btReadLine: (double) *timeout* error:(NSError ∗∗) *error*

Tries to read string data, ending with CR/LF up to specifed timeout.

**Note**

You can use bluethooth streams instead

**Parameters**

| | |
|---|---|
| *timeout* | maximim timeout in seconds to wait for data |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

string with the line read (can be empty string too) if function succeeded, nil otherwise

#### 1.7.2.18 - (BOOL) btSetDataNotificationMaxTime: (double) *maxTime* maxLength:(int) *maxLength* sequenceData:(NSData ∗) *sequenceData* error:(NSError ∗∗) *error*

Sets the conditions to fire the NSStreamEventHasBytesAvailable event on bluetooth streams.

If all special conditions are disabled, then the notification will be fired the moment data arrives. You can have multiple notifications active at the same time, for example maxBytes and maxTime.

**Parameters**

| | |
|---|---|
| *maxTime* | notification will be fired 'maxTime' seconds after the last byte arrives, passing 0 disables it. For example 0.1 means that 100ms after the last byte is received the notification will fire. |
| *maxLength* | notification will be fired after 'maxLength' data arrives, passing 0 disables it. |
| *sequenceData* | notification will be fired if the received data contains 'sequenceData', passing nil disables it. |

#### 1.7.2.19 - (BOOL) btSetEnabled: (BOOL) *enabled* error:(NSError ∗∗) *error*

Enables or disables bluetooth module.

Disabling the bluetooth module is currently the way to break existing bluetooth connection.

**Note**

When bluetooth module is enabled, access to the barcode engine is not possible!

**Parameters**

| | |
|---|---|
| *enabled* | TRUE to enable the engine, FALSE to disable it |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.7.2.20 - (BOOL) btWrite: (NSString ∗) *data* error:(NSError ∗∗) *error*

Sends data to the connected remote device.

**Note**

You can use bluethooth streams instead

**Parameters**

| | |
|---|---|
| *data* | data string to write |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.7.2.21 - (BOOL) btWrite: (void ∗) *data* length:(int) *length* error:(NSError ∗∗) *error*

Sends data to the connected remote device.

**Note**

You can use bluethooth streams instead

**Parameters**

| | |
|---|---|
| *data* | data bytes to write |
| *length* | the length of the data in the buffer |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

### 1.7.3 Properties

#### 1.7.3.1 - (NSInputStream∗) btInputStream `[read],[atomic],[assign]`

Bluetooth input stream, you can use it after connecting with btConnect.

See NSInputStream documentation.

#### 1.7.3.2 - (NSOutputStream*) btOutputStream [read], [atomic], [assign]

Bluetooth output stream, you can use it after connecting with btConnect.

See NSOutputStream documentation.

## 1.8 External Serial Port Functions

Functions to work with Linea Tab's external serial port.

### Functions

- (BOOL) - Linea::extOpenSerialPort:baudRate:parity:dataBits:stopBits:flowControl:error:

  *Opens the external serial port with specified settings.*
- (BOOL) - Linea::extCloseSerialPort:error:

  *Closes the external serial port opened with extOpenSerialPort.*
- (BOOL) - Linea::extWriteSerialPort:data:error:

  *Sends data to the connected remote device via serial port.*
- (NSData ∗) - Linea::extReadSerialPort:length:timeout:error:

  *Reads data from the connected remote device via serial port.*

### 1.8.1 Detailed Description

Functions to work with Linea Tab's external serial port.

### 1.8.2 Function Documentation

#### 1.8.2.1 - (BOOL) extCloseSerialPort: (int) *port* error:(NSError ∗∗) *error*

Closes the external serial port opened with extOpenSerialPort.

**Parameters**

| | |
|---|---|
| *port* | the port number, currently only 1 is used |
| *error* | returns error information, you can pass nil if you don't want it |

**Returns**

TRUE upon success, FALSE otherwise

#### 1.8.2.2 - (BOOL) extOpenSerialPort: (int) *port* baudRate:(int) *baudRate* parity:(int) *parity* dataBits:(int) *dataBits* stopBits:(int) *stopBits* flowControl:(int) *flowControl* error:(NSError ∗∗) *error*

Opens the external serial port with specified settings.

**Note**

On Linea Tab opening the serial port disables barcode scanner for the duration

**Parameters**

| | |
|---|---|
| *port* | the port number, currently only 1 is used |
| *baudRate* | serial baud rate |
| *parity* | serial parity, one of the PARITY_∗ constants (currenty only PARITY_NONE is supported) |
| *dataBits* | serial data bits, one of the DATABITS_∗ constants (currently only DATABITS_8 is supported) |
| *stopBits* | serial stop bits, one of the STOPBITS_∗ constants (currently only STOPBITS_1 is supported) |
| *flowControl* | serial flow control, one of the FLOW_∗ constants (currently only FLOW_NONE is supported) |
| *error* | returns error information, you can pass nil if you don't want it |

**Returns**

TRUE upon success, FALSE otherwise

#### 1.8.2.3 - (NSData ∗) extReadSerialPort: (int) *port* length:(int) *length* timeout:(double) *timeout* error:(NSError ∗∗) *error*

Reads data from the connected remote device via serial port.

**Parameters**

| | |
|---|---|
| *port* | the port number, currently only 1 is used |
| *length* | the maximum amount of data to read |
| *timeout* | timeout in seconds, passing 0 reads and returns the bytes currently in the buffer |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

NSData with bytes received if function succeeded, nil otherwise

#### 1.8.2.4 - (BOOL) extWriteSerialPort: (int) *port* data:(NSData ∗) *data* error:(NSError ∗∗) *error*

Sends data to the connected remote device via serial port.

**Parameters**

| | |
|---|---|
| *port* | the port number, currently only 1 is used |
| *data* | data bytes to write |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

## 1.9 Encrypted Magnetic Head Functions

Functions to work with Linea's optional encrypted magnetic head.

### Macros

- #define **LN_EMSR_EBASE** -11000
- #define LN_EMSR_EINVALID_COMMAND LN_EMSR_EBASE-0x01

  *Encrypted magnetic head invalid command sent.*
- #define LN_EMSR_ENO_PERMISSION LN_EMSR_EBASE-0x02

  *Encrypted magnetic head no permission error.*
- #define LN_EMSR_ECARD LN_EMSR_EBASE-0x03

  *Encrypted magnetic head card error.*
- #define LN_EMSR_ESYNTAX LN_EMSR_EBASE-0x04

  *Encrypted magnetic head command syntax error.*
- #define LN_EMSR_ENO_RESPONSE LN_EMSR_EBASE-0x05

  *Encrypted magnetic head command no response from the magnetic chip.*
- #define LN_EMSR_ENO_DATA LN_EMSR_EBASE-0x06

  *Encrypted magnetic head no data available.*
- #define LN_EMSR_EINVALID_LENGTH LN_EMSR_EBASE-0x14

  *Encrypted magnetic head invalid data length.*
- #define LN_EMSR_ETAMPERED LN_EMSR_EBASE-0x15

  *Encrypted magnetic head is tampered.*
- #define LN_EMSR_EINVALID_SIGNATURE LN_EMSR_EBASE-0x16

  *Encrypted magnetic head invalid signature.*
- #define LN_EMSR_EHARDWARE LN_EMSR_EBASE-0x17

  *Encrypted magnetic head hardware failure.*

### Functions

- (NSDictionary ∗) - Linea::emsrGetFirmwareInformation:error:

  *Returns information about the specified head firmware data.*
- (BOOL) - Linea::emsrIsTampered:error:

  *Checks if the head was tampered or not.*
- (BOOL) - Linea::emsrGetKeyVersion:keyVersion:error:

  *Retrieves the key version (if any) of a loaded key.*
- (BOOL) - Linea::emsrLoadInitialKey:error:

  *Loads Terminal Master Key (TMK) or reenable after tampering.*
- (BOOL) - Linea::emsrLoadKey:error:

  *Loads new key, in plain or encrypted with already loaded AES256 Key Encryption Key (KEK).*
- (NSData ∗) - Linea::emsrGetDUKPTSerial:

  *Returns DUKPT serial number, if DUKPT key is set.*
- (NSString ∗) - Linea::emsrGetDeviceModel:

  *Returns head's model.*
- (BOOL) - Linea::emsrGetFirmwareVersion:error:

  *Returns head's firmware version as number MAJOR∗100+MINOR, i.e.*
- (BOOL) - Linea::emsrGetSecurityVersion:error:

  *Returns head's security version as number MAJOR∗100+MINOR, i.e.*
- (NSData ∗) - Linea::emsrGetSerialNumber:

  *Return head's unique serial number as byte array.*

- (BOOL) - Linea::emsrUpdateFirmware:error:

  *Performs firmware update on the encrypted head.*
- (NSArray ∗) - Linea::emsrGetSupportedEncryptions:

  *Returns supported encryption algorhtms by the encrypted head.*
- (BOOL) - Linea::emsrSetEncryption:params:error:

  *Selects the prefered encryption algorithm.*
- (BOOL) - Linea::emsrConfigMaskedDataShowExpiration:unmaskedDigitsAtStart:unmaskedDigitsAtEnd-:error:

  *Fine-tunes which part of the card data will be masked, and which will be sent in clear text for display/print purposes.*

### 1.9.1 Detailed Description

Functions to work with Linea's optional encrypted magnetic head.

### 1.9.2 Macro Definition Documentation

#### 1.9.2.1 #define LN_EMSR_ECARD LN_EMSR_EBASE-0x03

Encrypted magnetic head card error.

#### 1.9.2.2 #define LN_EMSR_EHARDWARE LN_EMSR_EBASE-0x17

Encrypted magnetic head hardware failure.

#### 1.9.2.3 #define LN_EMSR_EINVALID_COMMAND LN_EMSR_EBASE-0x01

Encrypted magnetic head invalid command sent.

#### 1.9.2.4 #define LN_EMSR_EINVALID_LENGTH LN_EMSR_EBASE-0x14

Encrypted magnetic head invalid data length.

#### 1.9.2.5 #define LN_EMSR_EINVALID_SIGNATURE LN_EMSR_EBASE-0x16

Encrypted magnetic head invalid signature.

#### 1.9.2.6 #define LN_EMSR_ENO_DATA LN_EMSR_EBASE-0x06

Encrypted magnetic head no data available.

#### 1.9.2.7 #define LN_EMSR_ENO_PERMISSION LN_EMSR_EBASE-0x02

Encrypted magnetic head no permission error.

#### 1.9.2.8 #define LN_EMSR_ENO_RESPONSE LN_EMSR_EBASE-0x05

Encrypted magnetic head command no response from the magnetic chip.

#### 1.9.2.9 #define LN_EMSR_ESYNTAX LN_EMSR_EBASE-0x04

Encrypted magnetic head command syntax error.

#### 1.9.2.10 #define LN_EMSR_ETAMPERED LN_EMSR_EBASE-0x15

Encrypted magnetic head is tampered.

### 1.9.3 Function Documentation

#### 1.9.3.1 - (BOOL) emsrConfigMaskedDataShowExpiration: (BOOL) *showExpiration* unmaskedDigitsAtStart:(int) *unmaskedDigitsAtStart* unmaskedDigitsAtEnd:(int) *unmaskedDigitsAtEnd* error:(NSError ∗∗) *error*

Fine-tunes which part of the card data will be masked, and which will be sent in clear text for display/print purposes.

**Parameters**

| | |
|---|---|
| *showExpiration* | if set to TRUE, expiration date will be shown in clear text, otherwise will be masked |
| *unmaskedDigitsAtStart* | the number of digits to show in clear text at the start of the PAN, range from 0 to 6 (default is 4) |
| *unmaskedDigitsAtEnd* | the number of digits to show in clear text at the end of the PAN, range from 0, to 4 (default is 4) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.9.3.2 - (NSString ∗) emsrGetDeviceModel: (NSError ∗∗) *error*

Returns head's model.

**Returns**

head's model as string

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.9.3.3 - (NSData ∗) emsrGetDUKPTSerial: (NSError ∗∗) *error*

Returns DUKPT serial number, if DUKPT key is set.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

serial number or nil if an error occured

#### 1.9.3.4 - (NSDictionary ∗) emsrGetFirmwareInformation: (NSData ∗) *data* error:(NSError ∗∗) *error*

Returns information about the specified head firmware data.

Based on it, and the current head's name and firmware version you can chose to update or not the head's firmware

**Parameters**

| | |
|---|---|
| *data* | - firmware data |

**Returns**

dictionary containing extracted data or nil if the data is invalid. Keys contained are:

| | |
|---|---|
| "deviceModel" | Head's model, for example "EMSR-DEA" |
| "firmwareRevision" | Firmware revision as string, for example 1.07 |
| "firmwareRevisionNumber" | Firmware revision as number MAJOR∗100+MINOR, i.e. 1.07 will be returned as 107 |

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.9.3.5 - (BOOL) emsrGetFirmwareVersion: (int ∗) *version* error:(NSError ∗∗) *error*

Returns head's firmware version as number MAJOR∗100+MINOR, i.e.

version 1.05 will be sent as 105

**Parameters**

| | |
|---|---|
| *version* | integer, where firmware version is stored upon success |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.9.3.6 - (BOOL) emsrGetKeyVersion: (int) *keyID* keyVersion:(int ∗) *keyVersion* error:(NSError ∗∗) *error*

Retrieves the key version (if any) of a loaded key.

**Parameters**

| | |
|---|---|
| *keyID* | the ID of the key to get the version, one of the KEY_∗ constants |
| *keyVersion* | - pointer to integer, where key version will be returned upon success. Key version can be 0. |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.9.3.7 - (BOOL) emsrGetSecurityVersion: (int ∗) *version* error:(NSError ∗∗) *error*

Returns head's security version as number MAJOR∗100+MINOR, i.e.

version 1.05 will be sent as 105. Security version is the version of the certificated security kernel.

**Parameters**

| | |
|---|---|
| *version* | integer, where firmware version is stored upon success |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.9.3.8 - (NSData ∗) emsrGetSerialNumber: (NSError ∗∗) *error*

Return head's unique serial number as byte array.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

serial number or nil if an error occured

#### 1.9.3.9 - (NSArray ∗) emsrGetSupportedEncryptions: (NSError ∗∗) *error*

Returns supported encryption algorhtms by the encrypted head.

**Parameters**

| | |
|---|---|
| *data* | firmware file data |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

an array of supported algorithms or nil if an error occured

#### 1.9.3.10 - (BOOL) emsrIsTampered: (BOOL *) *tampered* error:(NSError **) *error*

Checks if the head was tampered or not.

If the head's tamper protection have activated, the device should be sent to service for checks

**Returns**

true if the head was tampered and not operational

**Exceptions**

| | |
|---|---|
| *NSPortTimeoutException* | if there is no connection to Linea |
| *LineaModuleInactive-Exception* | if there is no connection to the encrypted head |

#### 1.9.3.11 - (BOOL) emsrLoadInitialKey: (NSData *) *keyData* error:(NSError **) *error*

Loads Terminal Master Key (TMK) or reenable after tampering.

This command is enabled only if the device is in tamper mode or there is no TMK key yet. If the command is executed in normal mode an error will be returned. To reenable the device after tampering the old TMK key must be passed as an argument. If the keys do not match error will be returned.

**Parameters**

| | |
|---|---|
| *keyData* | an array, that consists of:<br>• BLOCK IDENT - 1 byte, set to 0x29<br>• KEY ID - the ID of the key to set, put KEY_TMK_AES (0x10)<br>• KEY VERSION - the version of the key in high to low order, 4 bytes, cannot be 0<br>• KEY - the key data, 16 bytes<br>• HASH - SHA256 of the previous bytes (BLOCK IDENT, KEY ID, KEY VERSION and KEY) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.9.3.12 - (BOOL) emsrLoadKey: (NSData *) *keyData* error:(NSError **) *error*

Loads new key, in plain or encrypted with already loaded AES256 Key Encryption Key (KEK).

Plain text loading works only the first time the specified key is loaded and is recommended only in secure environment. For normal usage the new key should be encrypted with the Key Encryption Key (KEK). The command is unavailable if the device is tampred.

**Parameters**

| | |
|---|---|
| *keyData* | an array, that consists of:<br>• MAGIC NUMBER - (1 byte) 0x2b<br>• ENCRYPTION KEY ID - (1 byte) the ID of the already existing key, used to encrypt the new key data. Set it to KEY_EH_AES256_LOADING (0x02) if you want to set the key in encrypted state or 0xFF for plain state.<br>• KEY ID - (1 byte) the ID of the key to set, one of the KEY_ constants. The TMK cannot be changed with this command.<br>• KEY VERSION - (4 bytes) the version of the key in high to low order, 4 bytes, cannot be 0<br>• KEY - (variable) the key data, length depends on the key in question, AES256 keys are 32 bytes, DUKPT key is 16 bytes key, 10 bytes serial, 6 bytes for padding (zeroes)<br>• HASH - SHA256 of the previous bytes (MAGIC NUMBER, ENCRYPTION KEY ID, KEY ID, KEY VERSION, KEY) |

If using KEY_EH_AES256_LOADING, then KEY + HASH have to be put inside the packet encrypted with AES256 using key KEY_EH_AES256_LOADING. SHA256 is calculated on the unencrypted data. The head decrypts the data and then calculates and compares the hash. If the calculated SHA does not match the SHA sent with the command, the key excahnge is rejected and error is returned.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.9.3.13 - (BOOL) emsrSetEncryption: (int) *encryption* params:(NSData ∗) *params* error:(NSError ∗∗) *error*

Selects the prefered encryption algorithm.

When card is swiped, it will be encrypted by it and sent via magneticCardEncryptedData delegate

**Parameters**

| | |
|---|---|
| *encryption* | encryption algorhtm used, one o fthe ALG_∗ constants |
| *params* | optional algorithm parameters, currently no algorithm supports these |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.9.3.14 - (BOOL) emsrUpdateFirmware: (NSData ∗) *data* error:(NSError ∗∗) *error*

Performs firmware update on the encrypted head.

DO NOT INTERRUPT THE COMMUNICATION DURING THE FIRMWARE UPDATE!

**Parameters**

| | |
|---|---|
| *data* | firmware file data |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

## 1.10 RF Reader Functions

Functions to work with the Linea's built-in RF cards reader.

### Macros

- #define CARD_SUPPORT_TYPE_A 0x0001

  *ISO14443 Type A (Mifare) cards will be detected.*
- #define CARD_SUPPORT_TYPE_B 0x0002

  *ISO14443 Type B cards will be detected.*
- #define CARD_SUPPORT_FELICA 0x0004

  *Felica cards will be detected.*
- #define CARD_SUPPORT_NFC 0x0008

  *NFC cards will be detected.*
- #define CARD_SUPPORT_JEWEL 0x0010

  *Jewel cards will be detected.*
- #define CARD_SUPPORT_ISO15 0x0020

  *ISO15693 cards will be detected.*

### Functions

- (BOOL) - Linea::rfInit:error:

  *Initializes and powers on the RF card reader module.*
- (BOOL) - Linea::rfClose:

  *Powers down RF card reader module.*
- (BOOL) - Linea::rfRemoveCard:error:

  *Call this function once you are done with the card, a delegate call rfCardRemoved will be called when the card leaves the RF field and new card is ready to be detected.*
- (BOOL) - Linea::mfAuthByKey:type:address:key:error:

  *Authenticate mifare card block with direct key data.*
- (BOOL) - Linea::mfStoreKeyIndex:type:key:error:

  *Store key in the internal module memory for later use.*
- (BOOL) - Linea::mfAuthByStoredKey:type:address:keyIndex:error:

  *Authenticate mifare card block with previously stored key.*
- (NSData ∗) - Linea::mfRead:address:length:error:

  *Reads one more more blocks of data from Mifare Classic/Ultralight cards.*
- (int) - Linea::mfWrite:address:data:error:

  *Writes one more more blocks of data to Mifare Classic/Ultralight cards.*
- (BOOL) - Linea::mfUlcSetKey:key:error:

  *Sets the 3DES key of Mifare Ultralight C cards.*
- (BOOL) - Linea::mfUlcAuthByKey:key:error:

  *Performs 3DES authentication of Mifare Ultralight C card using the given key.*
- (NSData ∗) - Linea::iso15693Read:startBlock:length:error:

  *Reads one more more blocks of data from ISO 15693 card.*
- (int) - Linea::iso15693Write:startBlock:data:error:

  *Writes one more more blocks of data to ISO 15693 card.*
- (NSData ∗) - Linea::iso15693GetBlocksSecurityStatus:startBlock:nBlocks:error:

  *Reads the security status of one more more blocks from ISO 15693 card.*
- (BOOL) - Linea::iso15693LockBlock:block:error:

  *Locks a single ISO 15693 card block.*

- (BOOL) - Linea::iso15693WriteAFI:afi:error:

  *Changes ISO 15693 card AFI.*
- (BOOL) - Linea::iso15693LockAFI:error:

  *Locks ISO 15693 AFI preventing further changes.*
- (BOOL) - Linea::iso15693WriteDSFID:dsfid:error:

  *Changes ISO 15693 card DSFID.*
- (BOOL) - Linea::iso15693LockDSFID:error:

  *Locks ISO 15693 card DSFID preventing further changes.*
- (NSData ∗) - Linea::felicaRead:startBlock:length:error:

  *Reads one more more blocks of data from FeliCa card.*
- (int) - Linea::felicaWrite:startBlock:data:error:

  *Writes one more more blocks of data to FeliCa card.*
- (BOOL) - Linea::felicaSmartTagGetBatteryStatus:status:error:

  *Returns FeliCa SmartTag battery status.*
- (BOOL) - Linea::felicaSmartTagClearScreen:error:

  *Clears the screen of FeliCa SmartTag.*
- (BOOL) - Linea::felicaSmartTagDrawImage:image:topLeftX:topLeftY:drawMode:layout:error:

  *Draws image on FeliCa SmartTag's screen.*
- (BOOL) - Linea::felicaSmartTagSaveLayout:layout:error:

  *Saves the current display as layout number.*
- (BOOL) - Linea::felicaSmartTagDisplayLayout:layout:error:

  *Displays previously stored layout.*
- (int) - Linea::felicaSmartTagWrite:address:data:error:

  *Writes data in FeliCa SmartTag.*
- (NSData ∗) - Linea::felicaSmartTagRead:address:length:error:

  *Writes data in FeliCa SmartTag.*
- (BOOL) - Linea::felicaSmartTagWaitCompletion:error:

  *Waits for FeliCa SmartTag to complete current operation.*

### 1.10.1 Detailed Description

Functions to work with the Linea's built-in RF cards reader.

### 1.10.2 Macro Definition Documentation

#### 1.10.2.1 #define CARD_SUPPORT_JEWEL 0x0010

Jewel cards will be detected.

Currently unsupported.

#### 1.10.2.2 #define CARD_SUPPORT_NFC 0x0008

NFC cards will be detected.

Currently unsupported.

#### 1.10.2.3 #define CARD_SUPPORT_TYPE_B 0x0002

ISO14443 Type B cards will be detected.

Currently unsupported.

### 1.10.3 Function Documentation

#### 1.10.3.1 - (NSData ∗) felicaRead: (int) *cardIndex* startBlock:(int) *startBlock* length:(int) *length* error:(NSError ∗∗) *error*

Reads one more more blocks of data from FeliCa card.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *startBlock* | the starting block to read from |
| *length* | the number of bytes to read, this must be multiple of block size (can be taken from the card info that is coming with rfCardDetected call) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

NSData object containing the data received or nil if an error occured

#### 1.10.3.2 - (BOOL) felicaSmartTagClearScreen: (int) *cardIndex* error:(NSError ∗∗) *error*

Clears the screen of FeliCa SmartTag.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *status* | upon successful execution, battery status will be returned here, one of FELICA_SMARTTAG-_BATTERY_∗ constants |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.3 - (BOOL) felicaSmartTagDisplayLayout: (int) *cardIndex* layout:(int) *layout* error:(NSError ∗∗) *error*

Displays previously stored layout.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *layout* | layout index (1-12) of the previously stored image |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.4 - (BOOL) felicaSmartTagDrawImage: (int) *cardIndex* image:(UIImage ∗) *image* topLeftX:(int) *topLeftX* topLeftY:(int) *topLeftY* drawMode:(int) *drawMode* layout:(int) *layout* error:(NSError ∗∗) *error*

Draws image on FeliCa SmartTag's screen.

The screen is 200x96 pixels.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *image* | image to draw |
| *topLeftX* | - topleft X coordinate in pixels |
| *topLeftY* | - topleft Y coordinate in pixels |
| *drawMode* | draw mode, one of the FELICA_SMARTTAG_DRAW_∗ constants |
| *layout* | only used when drawMode is FELICA_SMARTTAG_DRAW_USE_LAYOUT, it specifies the index of the layout (1-12) of the previously stored image |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.5 - (BOOL) felicaSmartTagGetBatteryStatus: (int) *cardIndex* status:(int ∗) *status* error:(NSError ∗∗) *error*

Returns FeliCa SmartTag battery status.

**Note**

Call this function before any other SmartTag

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *status* | upon successful execution, battery status will be returned here, one of FELICA_SMARTTAG-_BATTERY_∗ constants |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.6 - (NSData ∗) felicaSmartTagRead: (int) *cardIndex* address:(int) *address* length:(int) *length* error:(NSError ∗∗) *error*

Writes data in FeliCa SmartTag.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *address* | the address of the card to read from, refer to SmartTag documentation |
| *length* | of the data to read, note that the data does not need to be aligned to block size |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

NSData object containing the data received or nil if an error occured

#### 1.10.3.7 - (BOOL) felicaSmartTagSaveLayout: (int) *cardIndex* layout:(int) *layout* error:(NSError **) *error*

Saves the current display as layout number.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *layout* | layout index (1-12) to which the currently displayed image will be saved |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.8 - (BOOL) felicaSmartTagWaitCompletion: (int) *cardIndex* error:(NSError **) *error*

Waits for FeliCa SmartTag to complete current operation.

Waiting is generally not needed, but needed in case for example drawing an image and then saving the layout, you need to wait for the image to be drawn. Write operation forces waiting internally.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.9 - (int) felicaSmartTagWrite: (int) *cardIndex* address:(int) *address* data:(NSData *) *data* error:(NSError **) *error*

Writes data in FeliCa SmartTag.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *address* | the address of the card to write to, refer to SmartTag documentation |
| *data* | data to write, note that the data does not need to be aligned to block size |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

number of bytes actually written or 0 if an error occured

#### 1.10.3.10 - (int) felicaWrite: (int) *cardIndex* startBlock:(int) *startBlock* data:(NSData *) *data* error:(NSError **) *error*

Writes one more more blocks of data to FeliCa card.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *startBlock* | the starting block to write to |
| *data* | the data to write, it must be multiple of block size (can be taken from the card info that is coming with rfCardDetected call) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

number of bytes actually written or 0 if an error occured

#### 1.10.3.11 - (NSData *) iso15693GetBlocksSecurityStatus: (int) *cardIndex* startBlock:(int) *startBlock* nBlocks:(int) *nBlocks* error:(NSError **) *error*

Reads the security status of one more more blocks from ISO 15693 card.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *startBlock* | the starting block to read from |
| *nBlocks* | the number of blocks to get the security status |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

NSData object containing the data received or nil if an error occured

#### 1.10.3.12 - (BOOL) iso15693LockAFI: (int) *cardIndex* error:(NSError **) *error*

Locks ISO 15693 AFI preventing further changes.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.13 - (BOOL) iso15693LockBlock: (int) *cardIndex* block:(int) *block* error:(NSError **) *error*

Locks a single ISO 15693 card block.

Locked blocks cannot be written upon anymore.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *block* | the block index to lock |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.14 - (BOOL) iso15693LockDSFID: (int) *cardIndex* error:(NSError **) *error*

Locks ISO 15693 card DSFID preventing further changes.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.15 - (NSData *) iso15693Read: (int) *cardIndex* startBlock:(int) *startBlock* length:(int) *length* error:(NSError **) *error*

Reads one more more blocks of data from ISO 15693 card.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *startBlock* | the starting block to read from |
| *length* | the number of bytes to read, this must be multiple of block size (can be taken from the card info that is coming with rfCardDetected call) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

NSData object containing the data received or nil if an error occured

#### 1.10.3.16 - (int) iso15693Write: (int) *cardIndex* startBlock:(int) *startBlock* data:(NSData *) *data* error:(NSError **) *error*

Writes one more more blocks of data to ISO 15693 card.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *startBlock* | the starting block to write to |
| *data* | the data to write, it must be multiple of block size (can be taken from the card info that is coming with rfCardDetected call) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

number of bytes actually written or 0 if an error occured

#### 1.10.3.17 - (BOOL) iso15693WriteAFI: (int) *cardIndex* afi:(uint8_t) *afi* error:(NSError **) *error*

Changes ISO 15693 card AFI.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *afi* | new AFI value |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.18 - (BOOL) iso15693WriteDSFID: (int) *cardIndex* dsfid:(uint8_t) *dsfid* error:(NSError **) *error*

Changes ISO 15693 card DSFID.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *dsfid* | new DSFID value |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.19 - (BOOL) mfAuthByKey: (int) *cardIndex* type:(char) *type* address:(int) *address* key:(NSData *) *key* error:(NSError **) *error*

Authenticate mifare card block with direct key data.

This is less secure method, as it requires the key to be present in the program, the prefered way is to store a key once in a secure environment and then authenticate using the stored key.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *type* | key type, either 'A' or 'B' |
| *address* | the address of the block to authenticate |
| *key* | 6 bytes key |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.20 - (BOOL) mfAuthByStoredKey: (int) *cardIndex* type:(char) *type* address:(int) *address* keyIndex:(int) *keyIndex* error:(NSError **) *error*

Authenticate mifare card block with previously stored key.

This the prefered method, as no key needs to reside in application.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *type* | key type, either 'A' or 'B' |
| *address* | the address of the block to authenticate |
| *keyIndex* | the index of the stored key, you can have up to 8 keys stored (0-7) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.21 - (NSData ∗) mfRead: (int) *cardIndex* address:(int) *address* length:(int) *length* error:(NSError ∗∗) *error*

Reads one more more blocks of data from Mifare Classic/Ultralight cards.

A single read operation gets 16 bytes of data, so you can pass 32 on length to read 2 blocks, etc

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *address* | the address of the block to read |
| *length* | the number of bytes to read, this must be multiple of block size (16 bytes) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

NSData object containing the data received or nil if an error occured

#### 1.10.3.22 - (BOOL) mfStoreKeyIndex: (int) *keyIndex* type:(char) *type* key:(NSData ∗) *key* error:(NSError ∗∗) *error*

Store key in the internal module memory for later use.

**Parameters**

| | |
|---|---|
| *keyIndex* | the index of the key, you can have up to 8 keys stored (0-7) |
| *type* | key type, either 'A' or 'B' |
| *key* | 6 bytes key |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.23 - (BOOL) mfUlcAuthByKey: (int) *cardIndex* key:(NSData ∗) *key* error:(NSError ∗∗) *error*

Performs 3DES authentication of Mifare Ultralight C card using the given key.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *key* | 16 bytes 3DES key to authenticate with |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.24 - (BOOL) mfUlcSetKey: (int) *cardIndex* key:(NSData ∗) *key* error:(NSError ∗∗) *error*

Sets the 3DES key of Mifare Ultralight C cards.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *key* | 16 bytes 3DES key to set |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.25 - (int) mfWrite: (int) *cardIndex* address:(int) *address* data:(NSData ∗) *data* error:(NSError ∗∗) *error*

Writes one more more blocks of data to Mifare Classic/Ultralight cards.

A single write operation stores 16 bytes of data, so you can pass 32 on length to write 2 blocks, etc

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *address* | the address of the block to write |
| *data* | the data to write, must be multiple of the block size (16 bytes) |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

number of bytes actually written or 0 if an error occured

#### 1.10.3.26 - (BOOL) rfClose: (NSError ∗∗) *error*

Powers down RF card reader module.

Call this function after you are done with the reader.

**Parameters**

| | |
|---|---|
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.27 - (BOOL) rfInit: (int) *supportedCards* error:(NSError ∗∗) *error*

Initializes and powers on the RF card reader module.

Call this function before any other RF card functions. The module power consumption is highly optimized, so it can be left on for extended periods of time.

**Parameters**

| | |
|---|---|
| *supportedCards* | any combination of CARD_SUPPORT_∗ flags to mark which card types to be active. Enable only cards you actually plan to work with, this has high implication on power usage and detection speed. |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

#### 1.10.3.28 - (BOOL) rfRemoveCard: (int) *cardIndex* error:(NSError ∗∗) *error*

Call this function once you are done with the card, a delegate call rfCardRemoved will be called when the card leaves the RF field and new card is ready to be detected.

**Parameters**

| | |
|---|---|
| *cardIndex* | the index of the card as sent by rfCardDetected delegate call |
| *error* | pointer to NSError object, where error information is stored in case function fails. You can pass nil if you don't want that information |

**Returns**

TRUE if function succeeded, FALSE otherwise

# Index