CARDPRESSO
more than an application

cardPresso MANUAL

XXS XS XM XL XXL

Advanced Script Command Reference

AVAILABLE FOR: Microsoft Windows

cardPresso

Ver 1.2.10

by cardPresso, Lda

MORE THAN AN APPLICATION

May 2015

Click here to get back to Implementation Table

Contents

Introduction

Introduction

cardPresso 2.0 advanced script command reference.

Here you will find the reference to the advanced commands used in cardPresso.

This document is a draft version. The information found in this document is still under internal review. The contents of this document may be subject to changes. The accuracy of the information provided is not guaranteed. cardPresso, Lda. holds no liability for the consequences of the usage of the information in this document.

For support requests, find us at:
Website: http:\\www.cardpresso.com
Support: support@cardpresso.com

Technology / API Implementation Table

Technology / API Implementation Table

TAG API's			Microsoft	HID	IDENTIVE/SCM		ACS	DUALi	CASTLES	GEM	ELATEC	RF IDEAS	ELICTYS	EWT	SPRING CARD
			PCSC	OK 53x1	SDI01X	CLOUD	ACR 38	DE - 620	EZ100PU	USB-TR CT30	TWN3 MULTI ISO	Pc Prox Plus	v 2.0	J68x	Crazy Writer
GENERIC TAG		UID	M S	M S	M S	M S					M S			M S	M S
		Security	M S	M S	M S	M S					M S			M S	M S
		Data	M S	M S	M S	M S					M S			M S	M S
		Config	M	M	M						M			M	M
		MAD													
		NFC													
MIFARE	Classic	UID	M S W	M S W	M S W	M S W		M S W			M S	M	M S W	M S W	M S W
		Security	M S W	M S W	M S W	M S W		M S W			M S		M S W	M S W	M S W
		Data	M S W	M S W	M S W	M S W		M S W			M S		M S	M S	M S
		Config	M W	M W	M W	M W		M W			M W		M W	M W	M W
		NFC													
		MAD	M W	M W	M W	M W					M		M W	M	M W
	Desfire	UID	M W	M W	M W	M W						M	M W		M W
		Security	M W	M W	M W	M W							M W		M W
		Data	M W	M W	M W	M W							M W		M W
		Config	M W	M W	M W	M W							M W		M W
		NFC	M W	M W	M W	M W							M W		M W
	Ultralight	UID	M	M	M	M						M			
		Security	M	M	M	M									
		Data	M	M	M	M									
		Config	M	M	M	M									
SLE	SLE44x2 SLE55x2	Security	M S	M S	M S	M S	M S		M S				M S		
		Data	M S	M S	M S	M S	M S		M S				M S		
		Config	M S	M S	M S	M S	M S		M S				M S		
ISO15693	ISO 15693	UID	M	M	M	M						M			
		Security	M	M	M	M									
		Data	M	M	M	M									
		Config	M	M	M										
		NFC													
HID	ISO PROX	UID	M	M								M			
		Security													

		Data													
		Config													
EM	410x	UID	M	M								M	M		
	4200	UID	M	M								M	M		
ST	SRTAG-D	UID	M	M	M	M									
		Security		M	M	M									
		Data		M	M	M									
		Config		M	M	M									
		NFC													

Objects

Objects

3.1.QExModule Module Reference

The **QExModule** is the base module from which other cardPresso modules derive. It also provides properties and methods to retrieve general information regarding the software and manage global configurations of other modules, properties and functions.

Properties:

The **QExModule** properties provide general information regarding the software and software logs. Additionally, the .settings property provides access to the Settings object and it's inherent methods.

See more: Settings.

Returns:	Property:
QString	.vendorName
QString	.productName
QString	.moduleName
QExSettings	.settings
QExEventLog	.log

Methods:

The **QtExModule** methods provide general information and configuration regarding the software settings, errors and basic software configuration.

Returns:	Method:
String	.resourcesLocation (*string* **section**)
String	.storageLocation (*Int* **location**)
DataBlock	.execAction (*String* **actionID**,*String* **scriptID**)
QObject	.createQObject(*String* **objectName**, *QScriptValue* **P1**,*QScriptValue* **P2**,QScriptValue **P3**)
QScriptValue	.createObject(*String* objectName, *QScriptValue* **P1**,*QScriptValue* **P2**,QScriptValue **P3**)
Void	setProperty(*String* **ID**,value)
QVariant	property(*String* **ID**)
Bool	sleep(*Int* **milliSeconds**)
Bool	abort (*String* **errorMessage**)
Void	setLastError(*QExError* **error**)
QExError	lastError();
Void	enableScriptExceptions(*Bool* **enabled**)
Bool	scriptExceptions();

3.1.1.Properties

Enter topic text here.

moduleName

QString **moduleName**

Retrieves the name of the module as a String.

productName

QString **productName**

Returns the software product name as a String.

vendorName

QString **vendorName**

Returns the software vendor name as a String.

settings

QExSettings **settings**

Returns the software settings as a Settings Object Object.

Read more: Settings Object

log

QExSettings **log**

Returns the software log as an EventLog Object.

Read more: EventLog Object.

3.1.2.Methods

resourcesLocation

String **resourcesLocation(***string* **section)**

Returns the module resources location as a String.

If no section is specified, the default value will be automatically used.

storageLocation

String **storageLocation(***Int* **location)**

Required params: location

Returns the current user's operating system general storage folders as a string.

Valid **location** values are:

0. Current User Desktop Folder
1. Current User Documents folder
2. System Fonts folder
3. Current User Start menu Program folder
4. Current User Music folder
5. Current User Videos folder
6. Current User Pictures folder
7. cardPresso local Encoding folder
8. Current User local folder
9. Current User local cardPresso Installation folder
10. Local Cache folder

setProperty

Bool **setProperty(***String* **property, *value*)**

Required params: time.

Configures the specified **property** with the provided **value**.

The following properties are available.

Property	Description	Value Data Type	Default Value
TIMEOUTS.POOLING.TAG	Time to wait for a tag to be placed.	Integer (miliseconds)	30000
DETECTEDTAGS.IGNOREDUPLICATEUIDS	Enables or disables duplicate UID detection.	Bool (True/False)	True

property

QVariant **property(***String* **id)**

Required params: id

Returns the value of the specified property.

sleep

Bool **sleep(***Int* ***milliSeconds*****)**

Required params: milliSeconds

Pauses the script at the point of the **sleep** method for the amount of time set in **milliSeconds** (ms). Returns 'true' if the method is executed. Returns false if the method is interrupted by the user.

abort

Bool **abort(***String* ***errorMessage*****)**

Required params: errorMessage

Interrupts the script at the point of the **abort** method and displays the specified **errorMessage** as an error message.

setLastError

Void **setLastError(***QExError* ***error*****)**

Required params: error

Configures the lastError with the Error Object provided by the user or returned by an error event.

Note:
If an expected error occurs during script execution that error will be set as the lastError Object.
Unexpected errors are not set as a lastError Object.
If scriptExceptions are enabled, if an error occurs, the script stops being executed an error message is returned and the error is set as the lastError.
If scriptExceptions are disabled, if an error occurs, the script keeps being executed and the error is set as the lastError.

lastError

QExError **lastError()**

Returns the Error Object currently set as the lastError by the setLastError method as an Error Object.

Note:
If an expected error occurs during script execution that error will be set as the lastError Object.
Unexpected errors are not set as a lastError Object.
If scriptExceptions are enabled, if an error occurs, the script stops being executed an error message is returned and the error is set as the lastError.
If scriptExceptions are disabled, if an error occurs, the script keeps being executed and the error is set as the lastError.

enableScriptExceptions

Void **enableScriptExceptions(***Bool* **enabled)**

Required params: enabled

Enables scriptExceptions if set to true 'true' and disables scriptExceptions if set to 'false'.

Note:
If an expected error occurs during script execution that error will be set as the lastError Object.
Unexpected errors are not set as a lastError Object.
If scriptExceptions are enabled, if an error occurs, the script stops being executed an error message is returned

and the error is set as the lastError.
If scriptExceptions are disabled, if an error occurs, the script keeps being executed and the error is set as the lastError.

scriptExceptions

Bool **scriptExceptions()**

Checks if script exceptions are enabled. Returns 'true' if enabled and 'false' if disabled.

See also: enableScriptExceptions.

3.2.QExSettings Settings Object Reference

The **QExSettings** Object provides access to the methods that allow us to manage the module settings and configurations, stored by default on the application configuration file (.conf).

Methods:

The **QtExSettings** Object provides the following methods:

Returns:	Method:
String	value(*String* **id**)
Bool	setValue(*String*, **id**,*String* **value**);
Bool	setDefaultValue(String, **id**,*String* **value**);
QExSettingsValue	settingsValue(*String* **id**, **SaveMode**);
QExSettings	section(*String* **id**)
KeysList	keys()

3.2.1.Methods

value

String **value**(*String* **id**)

Required params: id

Returns the **value** currently set on the application configuration file setting identified by the **id**.

setValue

Bool **setValue**(String, **id**,String **value**);

Required params: id, value

Adds a new setting to the application configuration file with the specified **value** identified by the user submitted **id**.

setDefaultValue

Bool **setDefaultValue**(String, **id**,QVariant **value**);

Required params: id, value

Adds a session default value to a setting (**id**) with the specified **value**. If the setting (**id**) is not defined in the application configuration file, this default value will be used.

settingsValue

QExSettingsValue **settingsValue**(String, **id**, **SaveMode**);

Required params: id, value

Instantiates a Settings Vlaue object with the specified **id** and **SaveMode**.

section

QExSettings **section**(String **id**)

Required params: id

Allows us to use a QExSettings object as a section. Providing access to the methods that allow us to add settings under that same section.

encoding.settings.section("section").section("subsection").setValue("settingmeta","valuemeta");

keys

KeysList **keys**()

Returns a list of the currently configured settings and values in a Settings Object in an array.

3.2.2.QExSettings Related Objecs

QExSettingsValue

The **SettingsValue** Object provides provides easier setting manipulation by allowing to create an instance of a specific setting in a unique object which saves itself in the application configuration file each time it's value is altered.

The QExSettingsValue object holds the ID of the setting, the value of the setting and the **SaveMode** of the object.

Available **SaveMode** values are:

SaveAlways: The setting is saved in the configuration file even if the value is not changed.

OnlySaveIfChanged: The setting is only saved in the configuration file only if the value is changed.

Methods:

The **SettingsValue** Object provides the following methods:

Returns:	Method:
String	value(string **id**)
Bool	setValue(*string* **id**, *string* **value**)

Methods

value

String **value**(*String* **id**)

Required params: id

Returns the **value** currently set on the Settings Value Object identified by the **id**.

setValue

Bool **setValue**(String, **id**,String **value**);

Required params: id, value

Sets the **value** of the Settings Value Object identified by the specified **id**.

3.3.QExEventLog EventLog Object Reference

The **QExEventLog** Object provides access to the methods that allow us to handle event logs.

Methods:

The **QExEventLog** Object provides the following methods:

Returns:	Method:
Void	log(LogLevel **level**,String **msg**,QExDataHash **data**,DataType **type**);
Void	logError(QExError **error**);
Void	logData(LogLevel **level**,String **dataID**,QExDataHash **data**, DataType **type**);
Void	logDebug(QVariant **object**,QVariant **function**,QVariant **description**,QVariant **parameters**,QExDataHash **data**,DataType **type**);
Void	logWarning(QVariant **object**,QVariant **function**,QVariant **description**,QVariant **parameters**,QExDataHash **data**,DataType **type**);
Void	addHandler(QExEventLogHandlerRef **handler**);

3.3.1.Methods

log

Void **log**(LogLevel **level**,String **msg**,QExDataHash **data**,DataType **type**)

Required params: level, msg

Logs a user specified message (**msg**) with the specified log level (**level**).

Specific data can also be logged with the **data** parameter.
A specific data type can also be specified with the **type** parameter.

Note:
If no **data** is specified, cardPresso will provide a value by default.
If no **type** is specified, cardPresso will use DATA_TYPE_NULL by default.
If a data **type** is specified , the **data** being handled must also be provided.

Valid **LogLevel** values:

LOG_OFF - The log is off.
LOG_DEBUG - The Debug log level.
LOG_WARNING - The Warning log level.
LOG_ERROR - The Error log level.
LOG_CRITICAL - The Critical log level.
LOG_FATAL - The Fatal log level.
LOG_MEMORY - The Memory log level.
(QExEventLog.h)

Valid **DataType** values:

DATA_TYPE_NULL - No data was supplied or data type is unknown.
DATA_TYPE_IN - The data is related to incoming data (received.
DATA_TYPE_OUT - The data is related to outgoing data (sent.
DATA_TYPE_ERROR - The data is related to an error that occured.
DATA_TYPE_USER - The data is user defined. Any program can use DATA_TYPE_USER + N to have different logic for different types of data.

logError

Void **logError**(QExError **error**)

Required params: error

Logs the specified error object.

3.4.QExError Error Object Reference

The **QExError** Object provides access to the methods that provide generic error information, regarding errors that have occurred. It also allows to export error information to external scripts.

Methods:

The **QExError** Object provides the following Properties and Methods:

Returns:	Property:

Bool	isSet
String	severity
String	description
String	toString
String	text

Returns:	**Method:**
String	addicionalInfo(*String* **key**)
Void	addAddicionalInfo(*String* **key**, *QVariant* **value**);
Void	reset()

3.4.1.QExError Properties

isSet

Bool **isSet**

Verifies if there is an error object currently set. Returns 'true' if set or 'false' if not set.

severity

String **severity**

Returns the severity of the error as a string.

description

String **description**

Returns the error description as a string.

toString

String **toString**

Returns the error as a string.

text

String **text**

Returns the error as a string.

3.4.2.QExError Methods

additionalInfo

String **additionalInfo**(*String* **key**)

Required params: key

Returns the available additional information identified by the **key** as a string.

Note:
Additional info may need to be previously added by using the method addAddicionalInfo.

addAddicionalInfo

Void **addAdditionalInfo**(*String* **key**, *String* **value**);

Required params: key, value

Allows the user to add a detailed description of the error object.
More than one additional info may be added per error object.
Each additional info entry must be identified by a unique **key**.

The additional info **value** of each **key** can then be return by using the addicionalInfo method.

reset

Void **reset**()

Resets the current error object.

3.5.QScriptValue ScriptData Object Reference

The **QScriptValue** Object provides access to the methods that provide generic data manipulation. It is prepared to handle the most commonly used types of data and perform conversion operations between them.

Methods:

The **QScriptValue** Object provides the following Properties and Methods:

Returns:	Method:
bool	isNull()
void	append (*QScriptValue* **string**)
QScriptValue	rightJustified(*int* **width**,*string* **fill**)
QScriptValue	leftJustified(*int* **width**,*string* **fill**)
QScriptValue	rightJustified(*int* **width**,*quint8* **fill**)
QScriptValue	leftJustified(*int* **width**,*quint8* **fill**)
QScriptValue	left(*int* **length**);
QScriptValue	right (*int* **length**)
QScriptValue	mid (*int* **pos**, *int* **len**)
int	size()
QScriptValue	toLower()
QScriptValue	toUpper()
QScriptValue	reversed()
QString	toHex()
QString	toBase64()
QString	toUtf8()
QString	toString()
qint64	toInt()
qint64	toUInt()
QScriptValue	toBool()
QByteArray	toByteArray()
QVariant	toVariant()
QScriptValue	fromHex(*string* **hex**)
QScriptValue	fromBase64 (*string* **base64**)
QScriptValue	fromString(*string* **data**)
void	replace(*QScriptValue* **before**, *QScriptValue* **after**);
bool	writeToFile (*string* **fileName**)
bool	writeToFileD (*string* **fileName**)

3.5.1.Methods

Enter topic text here.

isNull

Bool **isNull**()

Returns true if the ScriptData Object is null and false if the object is not null.

append

void **append (***QExScriptValue* **string)**

When working with strings, appends the user specified **string** to the ScriptData Object.

rightJustified

QScriptValue **rightJustified(***int* **width,***string* **fill)**

When working with strings, it transforms the string to the specified length (**width**). If the string is shorter than the specified width, the empty characters will be filled wit the character specified with the **fill** parameter.
Characters will be added to the end of the string.

leftJustified

QScriptValue **leftJustified(***int* **width,***string* **fill)**

When working with strings, it transforms the string to the specified length (**width**). If the string is shorter than the specified width, the empty characters will be filled wit the character specified with the **fill** parameter.
Characters will be added to the beginning of the string.

rightJustified

QScriptValue **rightJustified(***int* **width,***quint8* **fill)**

When working with bytes, it transforms the byte to the specified length (**width**). If the byte is shorter than the specified width, the empty bytes will be filled wit the byte specified with the **fill** parameter.
Fill bytes will be added to the end of the original byte.

leftJustified

QScriptValue **leftJustified(***int* **width,***quint8* **fill)**

When working with bytes, it transforms the byte to the specified length (**width**). If the byte is shorter than the specified width, the empty bytes will be filled wit the byte specified with the **fill** parameter.
Fill bytes will be added to the start of the original byte.

left

QScriptValue **left(***int* **length);**

When working with strings, it returns the characters in the length specified by the **length** parameter, starting from the left of the string.

Note:
Character count starts at the number 0.

right

QScriptValue **right (***int* **length)**

When working with strings, it returns the characters in the length specified by the **length** parameter, starting from the left of the string.

Note:
Character count starts at the number 0.

mid

QScriptValue **mid (***int* **pos,** *int* **length)**

When working with strings, it returns the characters in the length specified by the **length** parameter, starting from the position within the string, specified by the **pos** parameter.

Note:
Character count starts at the number 0.

size

int **size()**

Returns the size of the ScriptData Object as an integer.

toLower

QScriptValue **toLower()**

Returns the ScriptData Object in lowercase, and returns it as a ScriptData Object.

toUpper

QScriptValue **toUpper()**

Returns the ScriptData Object in uppercase, and returns it as a ScriptData Object.

reversed

QScriptValue **reversed()**

Returns the ScriptData Object reversed, and returns it as a ScriptData Object.

toHex

String **toHex()**

Converts the ScriptData Object to hexadecimal, and returns it as a string.

toBase64

String **toBase64()**

Converts the ScriptData Object to Base64, and returns it as a string.

toUtf8

String **toUtf8()**

Converts the ScriptData Object to Utf8, and returns it as a string.

toString

QScriptValue **toString()**

Converts the ScriptData Object to a string, and returns it as a string.

toInt

Quint64 **toInt()**

Converts the ScriptData Object to an integer, and returns it as q*uint64.*

toUInt

Quint64 **toUInt()**

Converts the ScriptData Object to a UInt, and returns it as q*uint64.*

toBool

QScriptValue **toBool()**

Converts the ScriptData Object to bool, and returns it as a ScriptData Object.

toByteArray

QByteArray **toByteArray()**

Converts the ScriptData Object to a ByteArray, and returns it as a QByteArray.

toVariant

QVariant **toVariant()**

Converts the ScriptData Object to a Variant, and returns it as a QVariant.

fromHex

QScriptValue **fromHex(***string* **hex)**

Converts the provided **hex** string to a hexadecimal value and returns it as a ScriptData Object.

fromBase64

QScriptValue **fromBase64(***string* **base64)**

Converts the provided **base64** string to a Base64 value and returns it as a ScriptData Object.

fromString

QScriptValue **fromString(***string* **string)**

Converts the provided **string** string to a string value and returns it as a ScriptData Object.

replace

void **replace(***QScriptValue* **before,** *QScriptValue* **after);**

Replaces the value of the ScriptData Object specified in the parameter **before**, with the value of the ScriptData Object specified in the parameter **after**.

writeToFile

bool **writeToFile (***string* **fileName)**

Writes the ScriptData Object value to the file specified by **fileName**.

Note:
The **filename** string must contain the full path to the file to be written.

Modules

Modules

Modules are the main building blocks of any cardPresso solution. We use the term 'solution' because each situation calls for a different solution, and in that sense cardPresso 2.0 was envisioned as a set of building blocks developed to be completely modular so they can be adapted to any situation.

Being modular means that the user gets to decide which parts of the program are needed and should be used, and which parts are not.

cardPresso 2.0 is built of several modules that work independently of each other. A specific module can be called to perform a specific set of functions, as opposed to loading the entire program when your working situation only requires a specific operation, such as printing or encoding.

4.1.Application

The **Application** module is the base of all the major application related objects.
It derives from the QExModule and inherits all it's methods.

The **Application** module provides the methods that allow us to create and customize the foundations upon which the application will be built.

Returns:	Method:
Bool	addModule(*QObject* **module**)
Bool	addModule(*QExModule* **module**)
Int	exec()
Void	quit()
Void	log(*int* **logLevel**,*String* **message**)
Void	debug(*String* **message**)
Void	print(*String* **message**)
Int	showMessage(*String* **errorMessage**, String **Buttons**)
Void	showError(*String* **error**)
QScriptValue	runScript(*String* **filename**)
Bool	runScripts()
QScriptValue	runScriptWizard(*String* **fileName**,*String* **requiredOption**)
Void	loadCSS(*QScriptValue* **styleSheet**)
Void	loadCSSFolder(*String* **styleSheetFolder**)
Bool	loadActions(*String* **Folder**)
String	applicationPath()
String	applicationFilePath()

4.1.1.addModule

Bool **addModule(***QObject* **module)**

Required params: module.

Loads the module from an existing QObject object providing access to all the methods from the respective module.

Returns 'true' if the operation was successful and 'false' if the module was unsuccessfully loaded.

4.1.2.addModule

Bool **addModule(***QExModule* **module)**

Required params: module.

Loads the module from an existing QExModule object providing access to all the methods from the respective module.

Returns 'true' if the operation was successful and 'false' if the module was unsuccessfully loaded.

4.1.3.exec

int **exec()**

Required params: none.

Runs the application with the currently configured settings.

4.1.4.quit

int **quit()**

Required params: none.

Quits the application.

4.1.5.log

Void **log(***int* **logLevel,***String* **message)**

Required params: logLeve, message.

Logs the specified user **message** in the specified **logLevel**.

app.log(1,"Test");,Error:TypeError: Result of expression 'app.log' [QExEventLog(name = "")] is not a function., Line:1

4.1.6.debug

Void **debug***(String* **message***)*

Required params: none.

Logs the user specified **message** to the output window.

4.1.7.print

Void **print***(String* **message***)*

Required params: none.

Prints the user specified **message** to the output window.

4.1.8.showMessage

Int **showMessage***(String* **errorMessage**, *String* **buttons** =QString?*)*

Required params: errorMessage.

Displays a pop-up alert window with the user specified **errorMessage** and specified **buttons**.

Multiple buttons can be defined using the " **|** " separator in the string.

The **buttons** parameter supports the following standard buttons:
PENDING TESTING

Ok
Open
Save
Cancel
Close

Discard

Apply

Reset

RestoreDefaults

Help

SaveAll

Yes

YesToAll

No

NoToAll

Abort

Retry

Ignore

NoButton

4.1.9.showError

Void **showError***(String* **error***)*

Required params: error.

Shows a pop-up error message with the user specified **error** message.

4.1.10.runScript

QScriptValue **runScript***(String* **filename***)*

Required params: filename.

Runs the script file specified in the **filename** parameter.

4.1.11.runScripts

Bool **runScripts**()

Required params: none.

Runs the scripts in the ----------
PROPERTY?

4.1.12.runScriptWizard

QScriptValue **runScriptWizard***(String* **fileName***,String* **requiredOption***)*

Required params: filename, requiredOption.

Runs the Script Wizard file specified by the **fileName** parameter and provides the required option in the **requiredOption** parameter.

4.1.13.loadCSS

Void **loadCSS***(QScriptValue* **styleSheet***)*

Required params: styleSheet.

Loads the CSS file specified by the **styleSheet** parameter.

4.1.14.loadCSSFolder

Void **loadCSSFolder***(String* **styleSheetFolder***)*

Required params: styleSheetFolder.

Loads the CSS files contained in the folder specified by the **styleSheetFolder** parameter.

4.1.15.loadActions

Bool **loadActions***(String* **folder***)*

Required params: styleSheet.

Loads the actions files contained in the folder specified by the **folder** parameter.

4.1.16.applicationPath

String **applicationPath**()

Required params: styleSheet.

Returns the application path as a string.

4.1.17.applicationFilePath

String **applicationFilePath**()

Required params: styleSheet.

Returns the application executable file path as a string.

4.2.User Interface

The **UI** module is the base module for all User Interface related objects.
It derives from the QExModule and inherits all it's methods.

The **UI** module provides the methods that allow to instantiate a window or a widget, which can then be used to build the user interface elements of the application.

To access the UI module methods the UI module must be loaded with the addModule method.

The following methods are available in the UI module:

Returns:	**Method:**
QScriptValue	createWidget(*String* **widgetFile**)
QScriptValue	createDialog(*String* **types**)

4.3.Security

4.3.1.login

Bool **login(**String **key,**QScriptValue **data)**

Required params: "key", "data".

Authenticates the current MIFARE Classic Tag with a KeyType and a KeyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

4.3.2.isValid

Bool **isValid()**

Returns "true" if the current Tag is successfully authenticated and returns "false" if the current is not authenticated.

4.4.cPEncoding encoding Object Reference:

The **cpEncoding** Object inherits from the QExModule and is the base of all encoding related objects, methods and properties, allowing the managing of both encoders and tags.

The **cpEncoding Module** provides the methods that allow us to instantiate an encoder as a cpEncoder Object, which will then be used to interact with both encoders and tags.

To instantiate an encoder as a cpEncoder Object is to create a connection between the cardPresso software and the physical encoder (or encoders) being used. This will in turn allow us to send and receive commands and data from the encoder trough cardPresso.

The following methods are available to create a cpEncoder instance:

Returns:	**Method:**
cpEncoder	encoder(*String* **encodername**)
cpEncoderList	encoders(*String* **types**)

Note:
By default, cardPresso creates an instance of the selected encoder as a cpEncoder Object with the name 'encoder'.

4.4.1.Methods

encoder

cpEncoder **encoder(***String* **encodername)**

Required params: encodername

Creates an instance of a single encoder, identified by the exact name of the encoder (**encodername**) specified by the user and provides access to the cpEncoder methods and properties.

Important:
The **encodername** parameter must be an exact match to the name of the encoder.

See also: cpEncoder

encoders

cpEncoderList **encoders(***String* **types)**

Required params: types

Enumerates and returns a cpEndoderList of the currently available encoders, of the specified **types**.
A cpEncoderList Object contains an enumeration of all the detected encoders as cpEncoder Objects.

Supported **types** are:
- "*CONTACT*"
- "*CONTACTLESS*"

Each detected encoder is instantiated as a cpEncoder object and inherits all the respective properties and methods.

Each cpEncoder object can be retrieved from the cpEncoderList and iterated to gain access to the respective properties and methods:

See also: cpEncoder

4.4.2.cpEncoder encoder Object Reference:

The **cpEncoder** Object derives from the cpEncoding Object and is a virtual representation of a physical encoder. This is used by cardPresso to interact with the encoder.
Every method or property used by a **cpEncoder** Object will reflect on the physical encoder it represents.

Provided are several properties and methods which are used to either get information specific to the encoder or perform operations with the encoder itself, which includes interacting with Tags.

The **cpEncoder Module** also provides the methods that allow us to instantiate a tag as a cpEncodeTag Object, which will then be used to perform operations with tags.

Properties:

The **cpEncoder** Object properties provide several methods which return general information specific to the encoder.

Returns:	**Property:**
String	name
String	vendor
String	model
QObject	apis

Methods:

The **cpEncoder** Object methods allow us to validate if the encoder is correctly instantiated and also allows us to instantiate a tag as a cpEncodeTag object.

Returns:	**Method:**
Bool	isValid()
EncodeTagList	tagsInRange(String **type**)
EncodeTag	waitForTag(String **type**)

Properties

name

String **name**

Returns the encoder name as a String.

vendor

String **vendor**

Returns the encoder vendor name as a String.

model

String **model**

Returns the encoder model name as a String.

apis

QObject **apis**

The api provides direct access to the encoder manufacturer API methods .

Methods

isValid

Bool **isValid()**

Required params: none.

Returns (Bool)True if the encoder is correctly instantiated or false if the encoder was not found.

waitForTag

EncodeTag **waitForTag(***String* **type)**

Required params: type

Returns and instantiates the first encoder tag object of the specified **type** detected by the encoder and provides access to the EncodeTag Object methods and properties.

Note:
By default, cardPresso will wait 30 seconds for a valid tag to be placed in the encoder.
If no valid tag is placed in time, a timeout error will occur.
A different timeout value can be configured for the Encoding module with the setProperty method.

Important:
Valid tag **type** values depend on the encoder in use. Ex: "Q5", "MIFARE.1K"...
If no tag is detected by the encoder in the configured tag detection time (see: setProperty) the script will be interrupted and a timeout error will be returned.
The default tag detection time value is 30000 (30 seconds) by default.

tagsInRange

EncodeTagList **tagsInRange(***String* **type)**

Required params: type

Instantiates and returns a list of tags of the specified **type**, detected by the current Encoder as an EncodeTagList Object.

This instantiates each detected Tag and returns them enumerated in an EncodeTagList Object.

Each tag is instantiated as an cpEncodeTag Object and inherits all the respective properties and methods.

A specific cpEncodeTag Object can be retrieved from the EncodeTagList and iterated to gain access to the respective properties and methods.

Note:
The setProperty method can be used to configure the tag detection time in miliseconds.
The setProperty method can be used to detect and ignore duplicate cards (by UID).

See also: cpEncodeTag Object.
See also: cpEncodeTag Object.

Important:
Valid tag **type** values are encoder specific. Ex: "Q5", "MIFARE.1K"...
If no tag **type** is specified, all tag types detected by the encoder will be enumerated in the EncodeTagList object.

4.4.3.cpEncodeTag tag Object Reference:

A Tag Object derives from the cpEncoder Object. It's a representation of a Tag successfully detected by the encoder/reader and instantiated by cardPresso.
It provides several properties and methods that allow access and manipulation of the data stored in the Tag.

Properties:

The **cpEncodeTag** Object properties provide several methods which to make use of all the features each Tag supports.

Returns:	Property:
String	.type
String	.description
QObject	.uid
QObject	.security
QObject	.data
QObject	.config

Methods:

Returns:	Method:
Bool	.isValid()

Generic Tag Properties

type

String **type**

Returns the Tag Type as a String.

description

String **description**

Returns the Tag Description as a String.

uid

isValid

Bool **.uid.isValid()**

Validades if the **.uid** property is implemented. Returns 'true' if it is implemented and 'false' if it's not implemented.

read

DataBlock **read()**

Returns the Tag UID as a QScriptValue object.

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successful or an error value if there was an error during the read process.

security

isValid

Bool **.security.isValid()**

Validades if the **.security** property is implemented. Returns 'true' if it is implemented and 'false' if it's not implemented.

login

Bool **.security.login(***DataBlock* **data)**

Required params: "data".

Authenticates the current Tag with a user specified **data** value. Returns (Bool) 'True' if successful or 'False' if

unsuccessful.

In the following example we will authenticate a MIFARE Classic Tag and if successfully authenticated we will log authentication result:

data

isValid

Bool **.data.isValid()**

Validades if the **.data** property is implemented. Returns 'true' if it is implemented and 'false' if it's not implemented.

size

Int **.data.size()**

Required params: none.

Retrieves the total size of the current Tag as an integer value.

In the following example we will compare the size of the data we want to write with the size of the Tag. If the data is smaller the the Tag size, we will then write the data:

read

QScriptValue **.data.read**(*Int* **offset**,*Int* **size**)

Required params: offset, size.

Retrieves the data of a Tag in the specified **offset** and with the specified **size**. The sector and block are automatically calculated. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
By default, if not set trough the parameters, the offset is 0 and the size is -1:
ex: myReadData = tag.data.read(); // default offset=0, default size=-1

If only the offset is set, the size defaults to -1:
ex: myReadData = tag.data.read(2); // from offset 2. size defaults to -1

To set a size, an offset must be set as well:
ex: myReadData = tag.data.read(2,4); // from offset 2. size 4.
Read methods contain an **.error** property which will return 'undefined' if the read was successfull or an error value if there was an error during the read process.

write

Bool **.data.write**(*Int* **offset**,QScriptValue **data**)

Required params: offset, data.

Writes the **data** defined by the user, in the specified **offset** in the tag as a QScriptValue. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

config

isValid

Bool **.conig.isValid()**

Validades if the **.uid** property is implemented. Returns 'true' if it is implemented and 'false' if it's not implemented.

size

Int **.config.size**()

Required params: none.

Retrieves the total size of the Tag Configuration Memory.

read

QScriptValue **.config.read**(*Int* **offset,***Int* **size**)

Available params: offset, size**.**

Retrieves the data from the Tag Configuration Memory in the specified **offset** and with the specified **size**. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successfull or an error value if there was an error during the read process.

write

Bool **.config.write**(*Int* **offset**,*QScriptValue* **data**)

Required params: offset, data.

Writes the **data** defined by the user as a QScriptValue Object, in the specified **offset** in the Tag Configuration Memory. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Generic Tag Methods

isValid

Bool .**isValid()**

Required params: none.

Returns (Bool)True if the Tag is correctly instantiated or false if no Tag was found.

Generic Tag Sample Script

```
//return a list of the currently available contactless encoders
encoders = encoding.encoders("CONTACTLESS");

// for each of  the detected encoders, run the following script:
for (var iEncoder in encoders) {

//return the current encoder and log it
   encoder = encoders[iEncoder];
   log("Detected Encoder : " + encoder.name); //log the Encoder name

//detect the Tag available in the encoder and log it
   tag = encoder.waitForTag();
   log("Detected Tag : " + tag.type); //log the Tag name

 //login the MIFARE Classic tag and verify if login was successful
   //first we return an authentication attempt so we can validate if it was successful.
     tagSecurityLogin = tag.security.login("A",data.fromHex("FFFFFFFFFFFF"));

   //now we check if the authentication was successful. this means the code within the following if condition will only run if successfully authenticated
   if (authenticated = "True") {

      //IMPORTANT: Failing to authenticate a Tag several times will result in locking the Tag permanently, rendering it useless.

      //Now that we're in the section of the script that will run if the authentication is successful, we have more Tag operations available.
      //We will first write some sample data onto the Tag main memory and log it.

      //write sample data in the Tag main memory, from the offset (bit) 32
      myWriteData = tag.data.write(32, "Some sample data");


      //Now that we have successfully written data onto the Tag we will attempt to read it from the Tag and log it as well.

      //read the data to confirm it it was correctly saved in the Tag and if we can read from the Tag.
      myReadData = tag.data.read(32, 16);
}
}
```

4.4.4.Contactless Tag Types

MIFARE®

MIFARE Classic and MIFARE DESFire are registered trademarks of NXP B.V. and are used under license.

MIFARE Classic®

Property:	Returns:	Method:
UID	*QScriptValue*	read()
UID	*Bool*	isValid()
Security	*Bool*	login(*String* **key**, *QScriptValue* **data**)
Security	*Bool*	isValid()
Data	*Int*	size()
Data	*QScriptValue*	readBlock(*Int* **sector**,Int **block**)
Data	*Bool*	writeBlock(*Int* **sector**,Int **block**, *String* **data**)
Data	*QScriptValue*	read(*Int* **offset**,*Int* **size**)
Data	*Bool*	write(*Int* **offset**,*String* **data**)
Data	*Bool*	formatMAD(*String* **keyB**, *String* **keyA**, *Bool* **allSectors**, *Int* **madVersion**)
Data	*Bool*	isMADConfigured (*Int* **madVersion**)
Data		getMADMultiApplication(*Int* **madVersion**)
Data		setMADMultiApplication(*String* **keyB**, *Bool* **multiApp**, *Int* **madVersion**)
Data		getMADCardHolderInfo(*Int* **madVersion**)
Data		setMADCardHolderInfo(String **keyB**, *Int* **sector**, *String* **givenName**, *String* **surname**, *String* **sex**, *String* **anyOtherInfo**, *Int* **madVersion**)
Data		removeMADCardPublisherInfo(*String* **keyB**, *Int* **madVersion**)
Data		getMADCardPublisherInfo(*Int* **madVersion**)
Data		setMADCardPublisherInfo(*String* **keyB**, *Int* **sector**, String **publisherInfo**, *Int* **madVersion**)
Data		getMADApplicationIdentifier(*Int* ***sector***, *Int* **madVersion**)
Data		setMADApplicationIdentifier(*String* **keyB**, *Int* **sector**, String **applicationCode**, String **functionClusterCode**, *Int* **madVersion**)
Data		isMADSectorFree(*Int* **sector**, *Int* **madVersion**)
Config	*Int*	size()
Config	*QScriptValue*	read(*Int* **offset**,Int **size**)
Config	*Bool*	write(Int **offset**,*String* **data**)

Config	*QScriptValue*	readTrailler(Int **sector**)
Config	*Bool*	writeTrailler(Int **sector**,*String* **data**)

UID

Bool **isValid()**

Returns "true" if the current UID property is implemented is valid and returns "false" if the current UID property is not implemented.

String **read()**

Required params: none.

Returns the Tag UID.

Data

Int **size()**

Required params: none.

Retrives the total size of the current MIFARE Classic Tag as an integer value.

QScriptValue **readBlock(**Int **sector,**Int **block)**

Required params: sector, block.

Retrieves the contents of a specific **block** in the MIFARE Classic Tag as a QScriptValue Object which inherits the respective properties and methods.

Bool **writeBlock**(Int **sector**,Int **block**, QScriptValue **data**)

Required params: sector, block, data.

Writes the **data** defined by the user in the specified **sector** and **block** in the MIFARE Classic Tag as a QScriptValue. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

QScriptValue **read**(Int **offset**,Int **size**)

Required params: offset, size.

Retrieves the data of a MIFARE Classic Tag in the specified **offset** and with the specified **size**. The sector and block are automatically calculated. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successfull or an error value if there was an error during the read process.

Bool **write**(Int **offset**,QScriptValue **data**)

Required params: offset, data.

Writes the **data** defined by the user, in the specified **offset** in the MIFARE Classic Tag as a QScriptValue. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **formatMAD**(*String* **keyB**, *String* **keyA**, *Bool* **allSectors**, *Int* **madVersion**);

Required params: keyB.

Formats a MIFARE Classic card in the MAD standard. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

If no **keyA** value is provided the default **keyA** will be used.
If no **allSectors** value is provided the default value is 'true'.
If no **madVersion** value is provided, the default value will be '0' (zero).

Bool **isMadConfigured**(*Int* **madVersion**);

Required params: none;

Checks if a MIFARE Classic card is currently formatted in the MAD standard. Returns (Bool) 'True' if formatted or 'False' if not formatted.

If no **madVersion** value is provided, the default value will be '1' (one).

Bool **getMADMultiApplication**(*Int* **madVersion**);

Required params: keyB;

Returns if the mutiple application setting is enabled or disabled. Returns (Bool) 'True' if enabled or 'False' if not disabled.

If no **madVersion** value is provided, the default value will be '1' (one).

Bool **setMADMultiApplication**(*String* **keyB**, *Bool* **multiApp**, *Int* **madVersion**);

Required params: keyB;

Enables or disables multi application configuration for the MAD. Returns (Bool) 'True' if successful or 'False' if not successful.

If no **madVersion** value is provided, the default value will be '1' (one).

QScriptValue **getMADCardHolderInfo**(*Int* **madVersion**);

Required params: none;

Returns the current card holder info from the MAD in a QScriptData object with properties for each element of the card holder info:
The **surname** property.
The **givenName** property.
The **sex** property.
The **anyOtherInfo** property.

If no **madVersion** value is provided, the default value will be '1' (one).

QScriptValue **setMADCardHolderInfo**(*String* **keyB**, *Int* **sector**, *String* **givenName**, *String* **surname**, *String* **sex**, *String* **anyOtherInfo**, *Int* **madVersion**);

Required params: keyB, sector, givenName

Writes the card holder info in the MAD tag. Returns the written card holder info in a QScriptData with several properties for each element of the card holder info:
The **surname** property.
The **givenName** property.
The **sex** property.
The **anyOtherInfo** property.

If no **madVersion** value is provided, the default value will be '1' (one).

Bool **removeMADCardPublisherInfo**(*String* **keyB**, *Int* **madVersion**);

Required params: keyB;

Removes the data from the card publisher info. Returns (Bool) 'True' if successful or 'False' if not unsuccessful.

If no **madVersion** value is provided, the default value will be '1' (one).

Bool **getMADCardPublisherInfo**(*Int* **madVersion**);

Required params: none.

Returns the current publisher info stored in the Tag. Returns (Bool) 'True' if successful or 'False' if not

unsuccessful.

If no **madVersion** value is provided, the default value will be '1' (one).

Bool **setMADCardPublisherInfo**(*String* **keyB**, *Int* **sector**, String **publisherInfo**, *Int* **madVersion**);
Required params: keyB, sector, publisherInfo;

Writes the provided publisher info in the specified sector. Returns (Bool) 'True' if successful or 'False' if not unsuccessful.

If no **madVersion** value is provided, the default value will be '1' (one).

QScriptValue **getMADApplicationIdentifier**(*Int* ***sector,*** *Int* **madVersion**);
Required params: sector.

Returns a QScriptValue with the application identifier from the specified sector. The application identifier is composed of two properties: the **applicationCode** property and the **functionClusterCode** property.

If no **madVersion** value is provided, the default value will be '1' (one).

Bool **setMADApplicationIdentifier**(*String* **keyB**, *Int* **sector**, String **applicationCode**, String **functionClusterCode**, *Int* **madVersion**);
Required params: sector, applicationCode, functionClusterCode.

Writes the provided application identifier data in the specified sector. Returns (Bool) 'True' if successful or 'False' if not unsuccessful.

If no **madVersion** value is provided, the default value will be '1' (one).

Bool **isMADSectorFree**(*Int* **sector**, *Int* **madVersion**);
Required params: sector;

Returns if the specified MAD sector is free. Returns (Bool) 'True' if enabled or 'False' if not disabled.

If no **madVersion** value is provided, the default value will be '1' (one).

Config

Int **size**()
Required params: none.

Retrieves the total size of the MIFARE Classic Tag Configuration Sectors.

QScriptValue **read**(Int **offset,**Int **size**)
Available params: offset, size**.**

Retrieves the data from a MIFARE Classic Tag Configuration Block in the specified **offset** and with the specified **size**. The sector and block are automatically calculated. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successful or an error value if there was an error during the read process.

Bool **write**(Int **offset**,QScriptValue **data**)
Required params: offset, data.

Writes the **data** defined by the user as a QScriptValue Object, in the specified **offset** in the MIFARE Classic Tag Configuration Block. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

QScriptValue **readTrailler(**Int **sector)**

Required params: sector.

Returns the Trailler data from the specified **sector** Trailler in the MIFARE Classic Tag as a QScriptValue Object, .

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successful or an error value if there was an error during the read process.

QScriptValue **writeTrailler(**Int **sector**,QScriptValue **data)**

Required params: sector, data.

Writes the **data** defined by the user as a QScriptValue Object in the specified **sector** Trailler block in the MIFARE Classic Tag .

apis.mifare Property

Provides methods to handle MIFARE Classic Tag API features.

The following MIFARE Classic Tag API methods are available:

Returns:	Method:
Int	tagSectors()
Int	blocksPerSector(*Int* **sector**)
Bool	login(*String* **key**,*QScriptValue* **data**)
Int	dataSize()
Int	configSize()
QScriptValue	readSectorBlock(*Int* **sector**,*Int* **block**)
Bool	writeSectorBlock(*Int* **sector**,Int **block**, *QScriptValue* **data**)
QScriptValue	read(*Int* **offset**,*Int* **size**)
Bool	write(*Int* **offset**,*QScriptValue* **data**)
QScriptValue	readSectorTrailler(*Int* **sector**)
QScriptValue	writeSectorTrailler(*Int* **sector**,*QScriptValue* **data**)

Int **.apis.mifare.tagSectors()**

Required params: offset, size.

Returns the total amount of sectors the current MIFARE Classic Tag contains.

Int **.apis.mifare.blocksPerSector(**Int **sector)**

Required params: offset, size.

Returns the total amount of blocks a specified **sector** contains.

Bool **.apis.mifare.login(**String **key,**QScriptValue **data)**

Required params: "keyType", "keyData".

Authenticates the current MIFARE Classic Tag with the specified **Key** and **Data**. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Int **.apis.mifare.dataSize()**

Required params: none.

Retrives the total size of the current MIFARE Classic Tag as an integer value.

Int **.apis.mifare.configSize**()

Required params: none.

Retrieves the total size of the MIFARE Classic Tag Configuration Sectors.

QScriptValue **.apis.readSectorBlock(**Int **sector,**Int **block)**

Required params: sector, block.

Retrieves the contents of a specific **block** in the MIFARE Classic Tag as a QScriptValue Object which inherits the respective properties and methods.
Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successful or an error value if there was an error during the read process.

Bool **.apis.writeSectorBlock**(Int **sector**,Int **block**, *QScriptValue* **data**)

Required params: sector, block, data.

Writes the **data** defined by the user in the specified **sector** and **block** in the MIFARE Classic Tag as a *QScriptValue*. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

QScriptValue **.apis.read**(Int **offset**,Int **size**)

Required params: offset, size.

Retrieves the data of a MIFARE Classic Tag in the specified **offset** and with the specified **size**. The sector and block are automatically calculated. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successful or an error value if there was an error during the read process.

Bool **.apis.write**(Int **offset**,*QScriptValue* **data**)

Required params: offset, size.

Writes the **data** defined by the user, in the specified **offset** in the MIFARE Classic Tag as a *QScriptValue*. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

QScriptValue **.apis.readSectorTrailler(**Int **sector)**

Required params: offset, size.

Returns the Trailler data from the specified **sector** Trailler in the MIFARE Classic Tag as a QScriptValue Object, .
Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successful or an error value if there was an error during the read process.

QScriptValue **.apis.writeSectorTrailler(**Int **sector**,QScriptValue **data)**

Required params: offset, size.

Writes the **data** defined by the user as a QScriptValue Object in the specified **sector** Trailler block in the MIFARE Classic Tag .

Mifare Classic MAD (API)

The MAD API provides methods to perform advanced operations on the MAD, specifically to manipulate cpMifareMADSettings objects.

The MAD API

Property	Returns:	Method:
apis.mad	*QScriptValue*	formatMAD
apis.mad	*cpMifareMADSettings*	readMADSettings
apis.mad	NDEFMessage	writeMADSettings

Bool **formatMAD**(*String* **keyB**, *String* **keyA**, *Bool* **allSectors**, *Int* **madVersion**);
Required params: keyB.

Formats a MIFARE Classic card in the MAD format. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

If not **keyA** value is provided the default **keyA** will be used.
If no **allSectors** value is provided the default value is 'true'.
If no **madVersion** value is provided, the default value will be '0' (zero).

cpMifareMADSettings **readMADSettings**(*Int* **madVersion**);
Required params: keyB.

Returns a cpMifareMADSettings object which contains the current MAD settings and provides access to the cpMifareMADSettings object methods that allow changes the MAD settings, which can the be saved to the MAD configuration sectors with the writeMADSettings method.

If no **madVersion** value is provided, the default value will be '1' (zero).

cpMifareMADSettings **readMADSettings**(*Int* **madVersion**);
Required params: keyB.

Writes a cpMifareMADSettings object to the MAD settings sectors.

If no **madVersion** value is provided, the default value will be '1' (zero).

The cpMifareMADSettings object contains the current MAD settings. Once instantiated, this object also provides access to methods that allow the manipulation of the current MAD settings.
The MAD settings can then be written to the tag, using the writeMADSettings method.

The cpMifareMADSettings object provides the following methods:

Returns:	Method:
Int	getMADVersion();
Bool	getMultiApplication();
Bool	setMultiApplication(*Bool* **multiApp**);
Bool	getMADAvailable();
Bool	setMADAvailable(*Bool* **madAvailable**);
QScriptData	getCardHolderInfo();
Bool	setCardHolderInfo(*String* **keyB**, *Int* **sector**, *String* **givenName**, *String* **surname**, *String* **sex**,

	String **anyOtherInfo**, *Int* **madVersion**);
Bool	removeCardPublisherInfo();
QScriptData	getCardPublisherInfo(Int madVersion);
Bool	setCardPublisherInfo(Int sector, String publisherInfo);
Int	getApplicationIdentifier(*Int* **sector**);
Bool	setApplicationIdentifier(*Int* **sector**, *Hex* **applicationCode**, *Hex* **functionClusterCode**);
Bool	isSectorFree(*Int* **sector**);

Bool **getMADVersion**();
Required params: none.

Returns the current MAD version configured in the tag info stored in the Tag.

Bool **getMultiApplication**();
Required params: none

Returns if the mutiple application setting is enabled or disabled. Returns (Bool) 'True' if enabled or 'False' if not disabled.

Bool **setMultiApplication**(*Bool* **multiApp**);
Required params: none.

Enables or disables multi application configuration for the MAD. Returns (Bool) 'True' if successful or 'False' if not successful.

Bool **getMADAvailable**();
Required params: none.

Returns if the MAD is available. Returns (Bool) 'True' if enabled or 'False' if not disabled.

Bool **getMADAvailable**(*Bool* **madAvailable**);
Required params: madAvailable

Sets if MAD is available or not available in the tag. Returns (Bool) 'True' if successful or 'False' if not successful.

QScriptValue **getCardHolderInfo**();
Required params: none;

Returns the current card holder info in a QScriptData object with properties for each element of the card holder info:
The **surname** property.
The **givenName** property.
The **sex** property.
The **anyOtherInfo** property.

QScriptValue **setCardHolderInfo**(*Int* **sector**, *String* **givenName**, *String* **surname**, *String* **sex**, *String* **anyOtherInfo**);
Required params: sector, givenName

Writes the card holder info in the MAD tag. Returns the current card holder info in a QScriptData with several properties for each element of teh card holder info:

Bool **removeCardPublisherInfo**();
Required params: none.

Removes the data from the card publisher info. Returns (Bool) 'True' if successful or 'False' if not unsuccessful.

Bool **getCardPublisherInfo**();

Required params: none.

Returns the current publisher info stored in the Tag.

Bool **setCardPublisherInfo**(*Int* **sector**, String **publisherInfo**);

Required params: sector, publisherInfo;

Writes the provided publisher info in the specified sector. Returns (Bool) 'True' if successful or 'False' if not unsuccessful.

QScriptValue **getApplicationIdentifier**(*Int* ***sector***);

Required params: sector.

Returns a QScriptValue with the application identifier from the specified sector. The application identifier is composed of two properties: the **applicationCode** property and the **functionClusterCode** property.

Bool **setApplicationIdentifier**(*Int* **sector**, String **applicationCode**, String **functionClusterCode**);

Required params: sector, applicationCode, functionClusterCode.

Writes the provided application identifier data in the specified sector. Returns (Bool) 'True' if successful or 'False' if not unsuccessful.

Bool **isSectorFree**(*Int* **sector**);

Required params: keyB;

Returns if the specified MAD sector is free. Returns (Bool) 'True' if enabled or 'False' if not disabled.

MIFARE Classic Sample Script

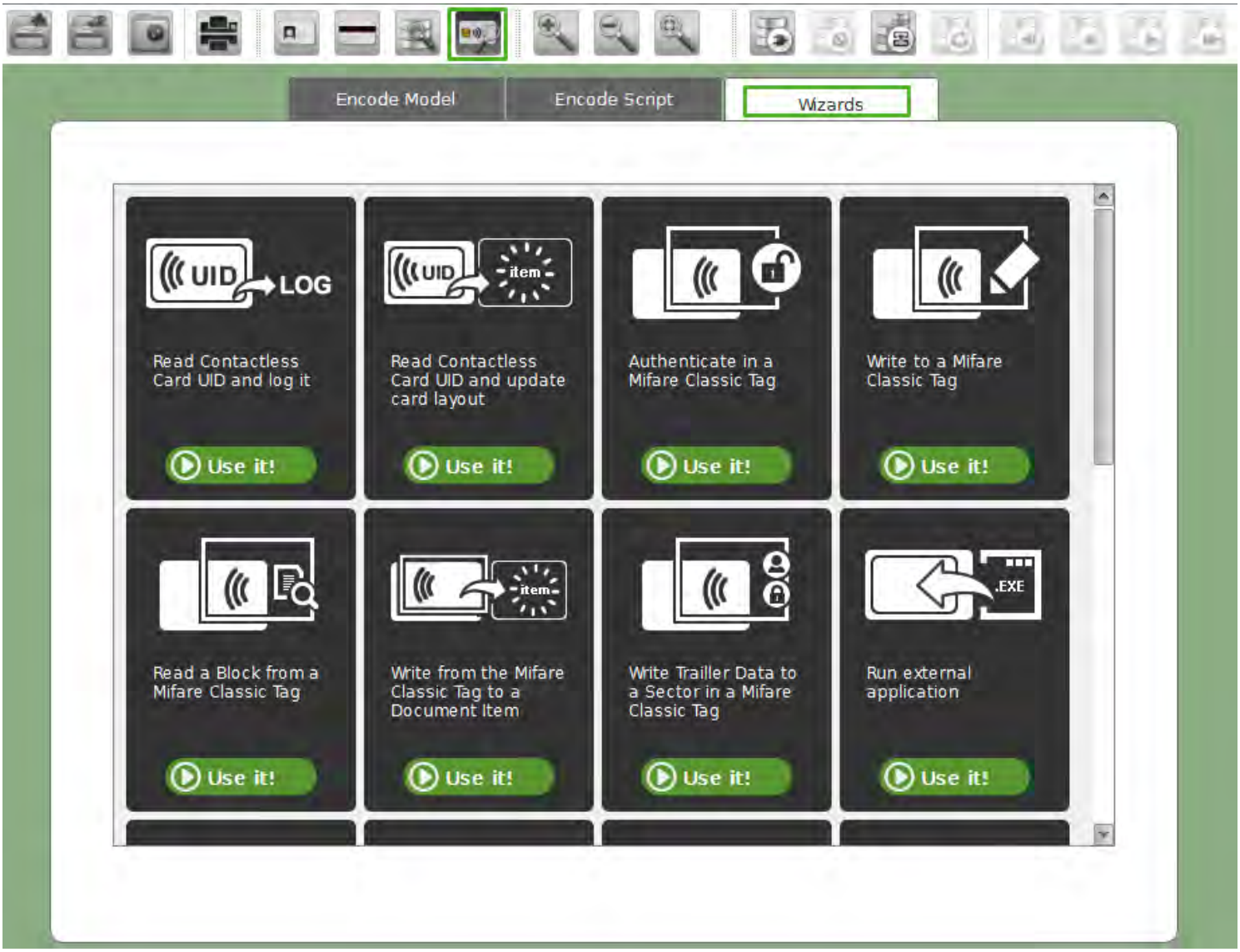
```
//return a list of the currently available contact encoders
encoders = encoding.encoders("Contactless");


// for each of  the detected encoders, run the following script:
for (var iEncoder in encoders) {

//return the current encoder and log it
   encoder = encoders[iEncoder];
   log("Detected Encoder : " + encoder.name); //log the Encoder name

   //detect the Tag available in the encoder and log it
   tag = encoder.waitForTag();
   log("Detected Tag : " + tag.type); //log the Tag name

   //login the MIFARE tag and verify if login was successful
   tagLogin = tag.security.login("A", data.fromHex("FFFFFFFFFFFF"));

   if (ltagLogin = "True") {

//IMPORTANT: Failing to authenticate a Tag several times will result in locking the Tag permanently, rendering it
useless.


      //Write sample data in the tag Sector 2, Block 2 and log it.
       myWriteData = tag.data.writeBlock(2,2 , "Some sample data");
      log("Written data: " + myWriteData);
```

```
    //read the data
    myReadData = tag.data.readBlock(2,2);


  }

}
```

MIFARE Classic Wizards

cardPresso has several wizards available to help you to build your script.

To access to Wizards menu, please click in the **Encode View » Wizards** tab.



MIFARE® DESFire®

Property	Returns:	Method:
UID	*Bool*	isValid()
UID	*String*	read()
Security	*Bool*	login(*Int* **keyNumber**, *String* **keyData**);
Security	*Bool*	loginISO(*Int* **keyNumber**, *String* **keyData**);
Security	*Bool*	loginISO24(*Int* **keyNumber**, *String* **keyData**);
Security	*Bool*	loginAES(*Int* **keyNumber**, *String* **keyData**);
Security	*Bool*	changeKey(*Int* **keyNumber**, *String* **newKeyData**);
Security	*Bool*	changeKeyISO(*Int* **keyNumber**, *String* **newKeyData**);
Security	*Bool*	changeKeyISO24(*Int* **keyNumber**, *String* **newKeyData**);
Security	*Bool*	changeKeyAES(*Int* **keyNumber**, *String* **newKeyData**);

Security	ScriptValue	keyDiversificationAES(String **masterKey**, Int **appId**=0,String **systemIdentifier**)
Security	ScriptValue	keyDiversificationAESAV1(String **masterKey**, Int **appId**=0)
Security	ScriptValue	keyDiversificationAESAV2(String **masterKey**, Int **appId**=0)
Security	ScriptValue	keyDiversification2TDEA(String **masterKey**, Int **appId**=0,String **systemIdentifier**, Int **masterKeyVersion**)
Security	ScriptValue	keyDiversification3TDEA(String **masterKey**, Int **appId**=0,String **systemIdentifier**, Int **masterKeyVersion**)
Data	*Bool*	isValid()
Data	*Int*	size()
Data	*Bool*	formatCard()
Data	*Bool*	createApplication(*Int* **appId**, *Int* **numberOfKeys**, *String* **encryptionType**, *Int* **isoId**, String **dfName**)
Data	*Bool*	deleteApplication(*Int* **appId**)
Data	*QScriptValue*	getApplicationIDS()
Data	*Int*	getFreeMemory()
Data	*QScriptValue*	getDFNames()
Data	*Bool*	selectApplication(*Int* **appId**)
Data	*Bool*	createFile(*Int* **fileId**, *String* **commMode**, *Int* **readKey**,*Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**, *Int* **fileSize**, *Int* **isoFileId**)
Data	*Bool*	createBackupFile(*Int* **fileId**, *Int* **commMode**, *Int* **readKey**,*Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**, *Int* **fileSize**, Int **isoFileId**)
Data	*Bool*	createValueFile(*Int* **fileId**, *String* **commMode**, *Int* **readKey**,*Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**,*Int* **lowerLimit**, Int **upperLimit**, *QScriptValue* **initialValue**,*Int* **limitedCreditEnabled**)
Data	*Bool*	createLinearRecordFile(*Int* **fileId**, *String* **commMode**, *Int* **readKey**, *Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**, *Int* **recordSize**, *Int* **maxRecords**, *Int* **isoFileId**)
Data	*Bool*	createCyclicRecordFile(*Int* **fileId**, *String* **commMode**, *Int* **readKey**, *Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**, *Int* **recordSize**, *Int* **maxRecords**, *Int* **isoFileId**)
Data	*Bool*	deleteFile(*Int* **fileId**)
Data	*QScriptValue*	getFileIDs()
Data	*QScriptValue*	getISOFileIDs()
Data	*Bool*	write(*Int* **fileId**, *Int* **offset**, *String* **data**)
Data	*Bool*	read(*Int* **fileId**, *Int* **offset**, *Int* **maxLength**)
Data	*QScriptValue*	getValue(*Int* **fileId**)
Data	*Bool*	credit(*Int* **fileId**, *Int* **amount**)
Data	*Bool*	debit(*Int* **fileId**, *Int* **amount**)
Data	*Bool*	limitedCredit(*Int* **fileId**, *Int* **amount**)
Data	*Bool*	writeRecord(*Int* **fileId**, *Int* **offset**, *String* **data**)

Data	*Bool*	readRecords(*Int* **fileId**, *Int* **fromRecord**, *Int* **maxRecordCount**, *Int* **recordSize**)
Data	*Bool*	clearRecordFile(*Int* **fileId**)
Data	*Bool*	commitTransaction()
Data	*Bool*	abortTransaction()
Data (NFC)	*QScriptValue*	formatNfc(*String* **masterKey**, *String* **nfcKey**)
Data (NFC)	*QScriptValue*	writeNfcText(*String* **nfcKey**, *String* **text**, *String* **locale**, *String* **encoding**)
Data (NFC)	*QScriptValue*	writeNfcUri(*String* **nfcKey**, *String,* **uri**)
Data (NFC)	*QScriptValue*	writeNfcIcon(*String* **nfcKey**, ScriptValue **icon**)
Data (NFC)	*QScriptValue*	writeNfcSmartPoster(*String* **nfcKey**, *String* **title**, *String* **uri**, *String* **action**, *String* **locale**, *String* **encoding**)
Data (NFC)	*Bool*	prepareNfcVCard()
Data (NFC)	*QScriptValue*	addNfcVCardField(*String* **field**, *String* **data**)
Data (NFC)	*QScriptValue*	writeNfcVCard(*String* **nfcKey**)
Data (NFC)	*QScriptValue*	lockNfcApplication(*String* **nfcKey**)
Config	*Bool*	isValid()
Config	*Bool*	setConfiguration(*Byte* **option**, *String* **data**) WARNING: Use with caution.
Config	*QScriptValue*	getKeySettings()
Config	*Bool*	changeKeySettings(*QScriptValue* **keySettings**)
Config	*QScriptValue*	getFileSettings(*Int* **fileId**)
Config	*Bool*	changeFileSettings(*Int* **fileId**, *String* **commMode**, *Int* **readKey**, *Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**)
Config	*QScriptValue*	getVersion()
Config	*QScriptValue*	getKeyVersion(*Int* **keyNumber**)

UID

Bool **isValid()**

Returns "true" if the current UID property is implemented is valid and returns "false" if the current UID property is not implemented.

String **read()**

Required params: none.

Returns the Tag UID if supported by the tag.

Bool **isValid()**

Returns "true" if the current Security property is implemented is valid and returns "false" if the current Security property is not implemented.

Bool **login**(*Int* **keyNumber**, *String* **keyData**);

Required params: keyNumber, keyData.

Authenticates in the currently selected application with the selected keyNumber and keyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **loginISO**(*Int* **keyNumber**, *String* **keyData**);

Required params: keyNumber, keyData.

Utilizes ISO authentication type to authenticate in the currently selected application with the selected keyNumber and keyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **loginISO24**(*Int* **keyNumber**, *String* **keyData**);

Required params: keyNumber, keyData.

Utilizes ISO24 authentication type to authenticate in the currently selected application with the selected keyNumber and keyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **loginAES**(*Int* **keyNumber**, *String* **keyData**);

Required params: keyNumber, keyData.

Utilizes AES authentication type to authenticate in the currently selected application with the selected keyNumber and keyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **changeKey**(*Int* **keyNumber**, *String* **keyData**);

Required params: keyNumber, keyData.

Changes the specified key of the currently selected application with the selected keyNumber and keyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **changeKeyISO**(*Int* **keyNumber**, *String* **keyData**);

Required params: keyNumber, keyData.

Changes the specified ISO key of the currently selected application with the selected keyNumber and keyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **changeKeyISO24**(*Int* **keyNumber**, *String* **keyData**);

Required params: keyNumber, keyData.

Changes the specified ISO24 key of the currently selected application with the selected keyNumber and keyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **changeKeyAES**(*Int* **keyNumber**, *String* **keyData , *Int* newkeyVersion**);

Required params: keyNumber, keyData.

Changes the specified AES key of the currently selected application with the selected keyNumber and keyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Note:
If no **newkeyVersion** is specified, the default value will be "0".

ScriptValue **keyDiversificationAES**(String **masterKey**, Int **appId**,String **systemIdentifier**)

Required params: masterKey, appId, systemIdentifier.

Uses a plain AES algorithm to return a **16 bytes AES 128 bits diversified key** or a **24 bytes AES 192 bits diversified key**, depending on the provided **masterKey**.

The diversification input size can range from 1 to 31 bytes.

The provided **masterKey** must be a 16 bytes AES 128 bits or a 24 bytes AES 192 bits.

Note:
If no **appID** is specified, the default value used will be "0".
If no **systemIdentifier** is specified, the default value will be provided by cardPreso.

ScriptValue **keyDiversificationAESAV1**(String **masterKey**, Int **appId**)

Required params: masterKey, appId.

Uses a plain AES algorithm to return a **16 bytes AES 128 bits diversified key** as performed in MIFARE SAM AV1 based on the provided **masterKey** and **appId**.

The provided **masterKey** must be 16 bytes AES 128 bits.

ScriptValue **keyDiversificationAESAV2**(String **masterKey**, Int **appId**)

Required params: masterKey, appId.

Uses an AES 128 CMAC based algorithm to return a **16 bytes AES 128 bits diversified key** as performed in MIFARE SAM AV2 based on the provided **masterKey** and **appId**.

The provided **masterKey** must be 16 bytes AES 128 bits.

ScriptValue **keyDiversification2TDEA**(String **masterKey**, Int **appId**,String **systemIdentifier,** Int **masterKeyVersion**)

Required params: masterKey, appId, systemIdentifier.

Returns a **16 bytes 2TDEA diversified key**.

The diversification input size can range from 1 to 15 bytes.

The provided **masterKey** must be 16 bytes 2TDEA.

Note:
If no **appID** is specified, the default value used will be "0".
If no **systemIdentifier** is specified, the default value will be provided by cardPreso.
If no **masterKey** is specified, the default value used will be "0".

ScriptValue **keyDiversification3TDEA**(String **masterKey**, Int **appId**,String **systemIdentifier,** Int **masterKeyVersion**)

Required params: masterKey, appId, systemIdentifier.

Returns a **24 bytes 3TDEA diversified key.**

The diversification input size can range from 1 to 15 bytes.

The provided **masterKey** must be 24 bytes 3TDEA.

Note:
If no **appID** is specified, the default value used will be "0".
If no **systemIdentifier** is specified, the default value will be provided by cardPreso.

If no **masterKey** is specified, the default value used will be "0".

Data

Bool **isValid()**

Returns "true" if the current Data property is implemented is valid and returns "false" if the current Data property is not implemented.

Int **size()**

Required params: none.

Retrives the total size of the current DESFire Tag as an integer value.

Bool **formatCard()**

Required params: none.

Resets the tag data memory to it's factory default removing all unprotected data from the tag data memory.

Note:
Must be authenticated with the master app key in application **0**.

Bool **createApplication**(*Int* **appId**, *Int* **numberOfKeys**, *String* **encryptionType**, *Int* **isoId**, String **dfName**)

Required params: appId, numberOfKeys.

Creates a new application in the current Tag with the specified app ID (**appId**) and number of keys (**numberOfKeys**).

Notes:
Each tag can have up to 28 applications.
If no **encriptionType** is specified, the application will have no encryption.
An application can have no keys if.
isoId and **dfName** are optional.
If no **isoId** is specified, the default value will be "**0**".
If no **dfName** is specified the default value will be an empty string.

Bool **deleteApplication**(*Int* **appId**)

Required params: appId, numberOfKeys.

Deletes the application in the current Tag with the specified app ID (**appId**).

QScriptValue ***g*etApplicationIDS**()

Required params: , .

Retrieves a list of the IDs of the existing applications in the current tag.

Note:
Must be authenticated with the master app key in application "**0**".

Int **getFreeMemory()**

Required params: none.

Retrives the total size of the currently free Data memory in the tag as an integer value.

QScriptValue **getDFNames**()

Required params: none.

Retrieves a list of the DF names of the existing application.

Note:
Must be authenticated with the master app key in application "**0**".

Bool **selectApplication**(*Int* **appId**)

Required params: none.

Selects the application specified by the **appId**.

Bool **createFile**(*Int* **fileId**, *String* **commMode**, *Int* **readKey**,*Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**, *Int* **fileSize**, *Int* **isoFileId**)

Required params: fileId, commMode, readKey, writeKey, readWriteKey, masterKey, fileSize.

Creates a new file in the currently selected and authenticated application.

Note:
If no **isoFileId** is specified, the default value will be "**0**".

Valid **commMode** values are:
"" = plain (empty)
"**PLAIN**" = Plain
"**PLAINSECURED**" = Maced
"**FULLSECURED**" = Enciphered
if no **commMode** is specified, Plain communication mode will be used.

Bool **createBackupFile**(*Int* **fileId**, *String* **commMode**, *Int* **readKey**,*Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**, *Int* **fileSize**, *Int* **isoFileId**)

Required params: fileId, commMode, readKey, writeKey, readWriteKey, masterKey, fileSize.

Creates a new Backup file in the currently selected and authenticated application.

Note:
Backup File changes are only saved or aborted when the commitTransaction and abortTransaction methods are executed.
If no **isoFileId** is specified, the default value will be "**0**".

Valid **commMode** values are:
"" = plain (empty)
"**PLAIN**" = Plain
"**PLAINSECURED**" = Maced
"**FULLSECURED**" = Enciphered
if no **commMode** is specified, Plain communication mode will be used.

Bool **createValueFile**(*Int* **fileId**, *String* **commMode**, *Int* **readKey**,*Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**, *Int* **lowerLimit**, Int **upperLimit**, *QScriptValue* **initialValue**,*Bool* **limitedCreditEnabled**)

Required params: fileId, commMode, readKey, writeKey, readWriteKey, masterKey, fileSize, lowerLimit, upperLimit, initialValue , limitedCreditEnabled.

Creates a new Value file in the currently selected and authenticated application.

Note:
Cyclic Record File changes are only saved or aborted when the commitTransaction and abortTransaction methods are executed.
The value for Limited Credit is limited to the sum of the debit commits, with the most recent transaction containing at least one debit.

Valid **commMode** values are:
"" = plain (empty)
"**PLAIN**" = Plain
"**PLAINSECURED**" = Maced
"**FULLSECURED**" = Enciphered
if no **commMode** is specified, Plain communication mode will be used.

Bool **createLinearRecordFile**(*Int* **fileId**, *String* **commMode**, *Int* **readKey**,*Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**, *Int* **recordSize**, *Int* **maxRecords**, *Int* **isoFileId**)
Required params: fileId, commMode, readKey, writeKey, readWriteKey, masterKey, recordSize, maxRecords.

Creates a new Linear Record file in the currently selected and authenticated application.

Note:
Linear Record File changes are only saved or aborted when the commitTransaction and abortTransaction methods are executed.
If no **isoFileId** is specified, the default value will be "**0**".

Valid **commMode** values are:
"" = plain (empty)
"**PLAIN**" = Plain
"**PLAINSECURED**" = Maced
"**FULLSECURED**" = Enciphered
if no **commMode** is specified, Plain communication mode will be used.

Bool **createCyclicRecordFile**(*Int* **fileId**, *String* **commMode**, *Int* **readKey**,*Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**, *Int* **recordSize**, *Int* **maxRecords**, *Int* **isoFileId**)
Required params: fileId, commMode, readKey, writeKey, readWriteKey, masterKey, recordSize, maxRecords.

Creates a new Cyclic Record file in the currently selected and authenticated application.

Note:
Cyclic Record File changes are only saved or aborted when the commitTransaction and abortTransaction methods are executed.
If no **isoFileId** is specified, the default value will be "**0**".

Valid **commMode** values are:
"" = plain (empty)
"**PLAIN**" = Plain
"**PLAINSECURED**" = Maced
"**FULLSECURED**" = Enciphered
if no **commMode** is specified, Plain communication mode will be used.

Bool **deleteFile**(*Int* **fileId**)
Required params: fileId.

Deletes the file specified by the **fileId** parameter in the currently authenticated application.

QScriptValue **getFileIDs**()

Required params: none.

Retrieves a list of existing file IDs in the currently authenticated application.

QScriptValue **getISOFileIDs**()

Required params: none.

Retrieves a list of ISO file IDs for the files in the currently authenticated application which have a specified ISO file ID.

Bool **write**(*Int* **fileId**, *Int* **offset**, *String* **data**)

Required params: fileId, offset, data.

Writes the **data** defined by the user, in the file specified by the **fileId** and the specified **offset** in the Tag. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Note:
This method is specific to regular files created by the createFile and createBackupFile methods.

Backup File changes are only saved or aborted when the commitTransaction and abortTransaction methods are executed.

QScriptData **read**(*Int* **fileId**, *Int* **offset**, *Int* **maxLength**)

Required params: fileId, offset, maxLength.

Retrieves the data from the file specified by the **fileId** in the specified **offset** and with the specified **maxLength**. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
This method is specific to regular files created by the createFile and createBackupFile method.

Backup File changes are only saved or aborted when the commitTransaction and abortTransaction methods are executed.

Read methods contain an **.error** property which will return 'undefined' if the read was successfull or an error value if there was an error during the read process.

QScriptData **getValue**(*Int* **fileId**)

Required params: fileId.

Retrieves the current value from the Value File specified by the **fileId**. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
This method is specific to Value files created by the createValueFile method.

Bool **credit**(*Int* **fileId**, *Int* **amount**)

Required params: fileId, amount.

Adds the value specified by the **amount** parameter to the Value File specified by the **fileId**. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Note:
This method is specific to Value files created by the createValueFile method.

Read methods contain an **.error** property which will return 'undefined' if the read was successfull or an error value if there was an error during the read process.

Bool **debit**(*Int* **fileId**, *Int* **amount**)

Required params: fileId, amount.

Subtracts the value specified by the **amount** parameter to the Value File specified by the **fileId**. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Note:
This method is specific to Value files created by the createValueFile method.

Read methods contain an **.error** property which will return 'undefined' if the read was successfull or an error value if there was an error during the read process.

Bool **limitedCredit**(*Int* **fileId**, *Int* **amount**)

Required params: fileId, amount.

Adds the value specified by the **amount** parameter to the Value File specified by the **fileId**. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Note:
The value for Limited Credit is **limited to the sum of the debit commits**, with the most recent transaction containing at least one debit.

This method is specific to Value files created by the createValueFile method.

Bool **writeRecord**(*Int* **fileId**, *Int* **offset**, *String* **data**)

Required params: fileId, offset, data.

Adds the **data** defined by the user, in the file specified by the **fileId** and the specified **offset** in the Record File. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Note:
If no commitTransaction command is sent after a writeRecord command, the next writeRecord command to the same file writes to the already created record.
After sending a commitTransaction command, a new writeRecord command will create a new record in the record file. An abortTransaction command will invalidate all changes.
This method is specific to regular files created by the createLinearRecordFile and createCyclicRecordFile methods.

QScriptData **readRecords**(*Int* **fileId**, *Int* **fromRecord**, *Int* **maxRecordCount**, *Int* **recordSize**)

Required params: fileId, fromRecord, maxRecordCount, recordSize.

Retrieves the **data**, in the file specified by the **fileId** and the specified **offset** in the Record File. Returns a QScriptData Object which inherits the respective properties and methods.

Note:
This method is specific to regular files created by the createLinearRecordFile and createCyclicRecordFile methods.

Bool **clearRecordFile**(*Int* **fileId**)

Required params: fileId.

Clears all the **data** in the file specified by the **fileId**. Returns a QScriptData Object which inherits the respective properties and methods. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Note:
This method is specific to regular files created by the createLinearRecordFile and createCyclicRecordFile methods.

Bool **commitTransaction**()

Required params: none.

Saves the the **data** changes on Backup Files. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Note:
This method is specific to regular files created by the createBackupFile methods.

Bool **abortTransaction**()

Required params: none.

Aborts the **data** changes on Backup Files. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Note:
This method is specific to regular files created by the createBackupFile methods.

Data (NFC)

Bool **formatNfc(**String **masterKey,**String **nfcKey)**

Required params: masterKey.

Configures the MIFARE DESFire card to support NFC data.

Note: This command creates an NFC application where the NFC data will be saved.

masterKey refers to the PICC master key.
nfckey refers to the key to be attributed to the NFC application
If no **nfcKey** is provided the PICC master key will be used.

QScriptValue **writeNfcText**(*String* **nfcKey**, *String* **text**, *String* **locale**, *String* **encoding**)

Required params: appKey, text.

Writes a plain text object with the user defined **text** to an NFC formatted MIFARE DESFire tag.

The **locale** parameter defines the language being used.
If no **locale** is provided the default locale will be "en-US"

The **encoding** parameter defines the character set being used.
If no **encoding** is provided the default locale will be "UTF8"
Supported charsets are **UTF8** and **UTF16**.

QScriptValue **writeNfcUri**(*String* **nfcKey**, *String* **uri**)

Required params: appKey, uri.

Writes a URI object with the user defined **uri** to an NFC formatted MIFARE DESFire tag.

QScriptValue **writeNfcIcon**(*String* **nfcKey**, *QScriptValue* **icon**)

Required params: appKey, icon.

Writes an **icon** object (image) to an NFC formatted MIFARE DESFire tag.

QScriptValue **writeNfcSmartPoster**(*String* **nfcKey**, *String* **title**, *String* **uri**, *String* **action**, *String* **locale**, *String* **encoding**)

Required params: appKey, icon.

Writes a smart poster object with the user defined **title** and **uri** to an NFC formatted MIFARE DESFire tag.

The **action** parameter suggests a course of action that the device (reader) should do with the content.

The following actions are supported:

DO: runs/opens the provided data (send an sms / open a browser / perform a telephone call).
SAVE: saves the provided data (store an sms / save the URI in a bookmark / save the telephone number in contacts).
EDIT: opens the provided data for editing (open an sms in an sms editor / open an URI in a URI editor / open the telephone number for editing).

Note: The action will only be executed if the device (reader) supports the provided data requested action.

Note:
The **locale** parameter defines the language being used.
If no **locale** is provided the default locale will be "en-US"

The **encoding** parameter defines the character set being used.
If no **encoding** is provided the default locale will be "UTF8"
Supported charsets are **UTF8** and **UTF16**.

QScriptValue **prepareNfcVCard**()

Required params: none.

Creates a new vCard record with several commonly used vCard Fields ready for usage.

New vCard Fields can be added with the addNfcVCardField method.

After editing, the writeNfcVCard command must be used, to save the vCard data in the tag.

The default VCard fields created by the **prepareNfcVCard** method are:

first_name	middle_name	family_name	prefix_name	suffix_name	nickname	email
email_alternative	pager	fax	home_phone	cell_phone	business_phone	pro_Address1
pro_Address2	pro_Post_Code	pro_Region_State	pro_Town	pro_Country	address1	address2
post_Code	region_State	town	country	birthday	photo	role
title	company	note				

QScriptValue **addNfcVCardField**(*String* **field**, *String* **data**)

Required params: field, data.

Adds a new field to an existing NFC vCard (see prepareNfcVCard).
The field is identified by the **field** parameter and will contain the data provided in the **data** parameter.

QScriptValue **writeNfcVCard**(*String* **nfcKey**)

Required params: nfcKey.

Saves the data from an existing NFC vCard object (see prepareNfcVCard) in the tag.

QScriptValue **lockNfcApplication**(*String* **nfcKey**)

Required params: nfcKey.

Locks the NFC Application, preventing further changes to the saved data.

Config

Bool **isValid()**

Returns "true" if the current Config property is implemented is valid and returns "false" if the current Data property is not implemented.

Bool **setConfiguration**(*Byte* **option**, *String* **data**)

Configures the specified tag **option** with the user specified configuration **data**. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

WARNING: Using this command can make changes in card permanent and possibly render it useless. Use with caution.

QScriptValue **getKeySettings()**

When authenticated in an application, retrieves a list of the existing keys and their current configuration.

Returns a QScriptValue with the following properties (case sensitive):
KeySettings
MaxNoOfkeys

Bool **changeKeySettings**(*QScriptValue* **keySettings**)

When authenticated in an application, allows the key settings to be modified with the user provided **keySettings**. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

QScriptValue **getFileSettings**(*Int* **fileId**)

When authenticated in an application, retrieves a QScriptValue wich contains all the settings for the key settings to be modified with the provided **keySettings**. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Returns a QScriptValue with the following properties:
FileType
CommMode
AccessRights
ReadKey
WriteKey
ReadWriteKey

MasterKey

Backup data files also contain the following properties:
FileSize

Value Files also contain the following properties:
LowerLimit
UpperLimit
LimitedCreditValue
LimitedCreditEnabled

Linear Record files and Cyclic Record files also contain the following properties
RecordSize
MaxNumberOfRecords
CurrentNumberOfRecords

Bool **changeFileSettings**(*Int* **fileId**, *String* **commMode**, *Int* **readKey**, *Int* **writeKey**, *Int* **readWriteKey**, *Int* **masterKey**)

When authenticated in an application, allows to change the settings in the available parameters. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

QScriptValue **getVersion**()

When authenticated in an application, returns a QScriptValue object with the MIFARE DESFire Software Version of the current tag (EV0/EV1).

The QScriptValue object also contains the the following properties:

hwVendorID (hardware vendor ID)
hwType (hardware type)
hwSubType (hardware sub type)
hwMajorVersion (hardware major version)
hwMinorVersion (hardware minor version)
hwStorageSize (hardware storage size)
hwProtocol (hardware protocol)
swVendorID (software vendor ID)
swType (software type)
swSubType (software sub type)
swMajorVersion (software major version)
swMinorVersion (software minor version)
swStorageSize (software storage size)
swProtocol (software protocol)
UID (tag UID)
batchNo (batch number)
CWProduction (Production CW)
YearProduction (production year)

QScriptValue **getKeyVersion**(*Int* **keyNumber**)

When authenticated in an application, returns the current version of the key as a QScriptValue object.

Note:
Only applies to keys with AES encryption who have been previously changed (changeKeyAES) and attributed a key version (**newkeyVersion**).

NFC MIFARE DESFire API

Property	Returns:	Method:
apis.nfc	*Bool*	SetNfcDESFireVersion(String **nfcDesfireVersion**)
apis.nfc	*String*	GetNfcDESFireVersion()

Bool **SetNfcDESFireVersion(**String **nfcDesfireVersion)**

Required params: nfcDesfireVersion.

Configures the NFC DESFire version to the user specified version.

Valid versions are **EV0** and **EV1**.

String **GetNfcDESFireVersion()**

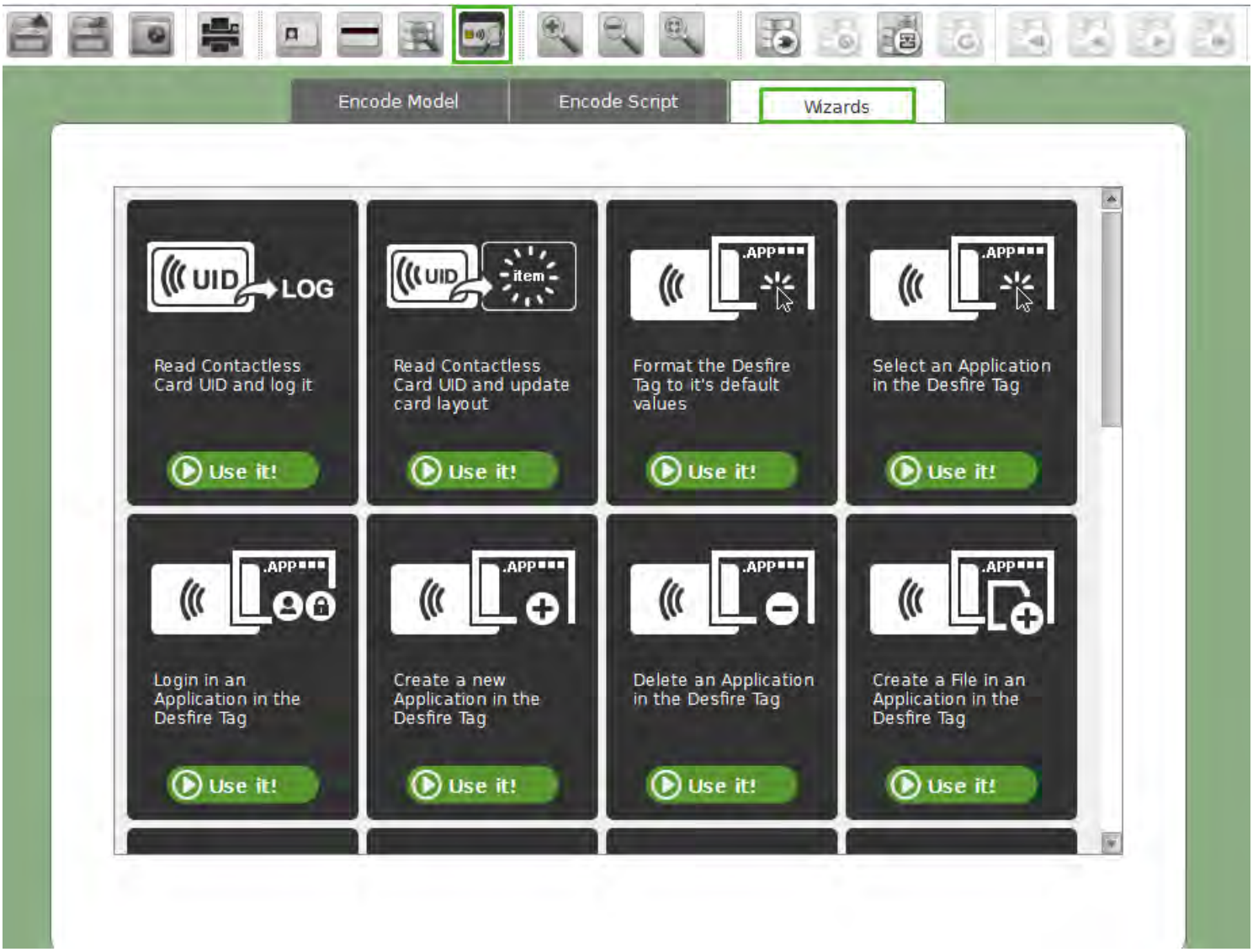
Required params: none.

Returns the currently configured NFC DESFire version.

Valid versions are **EV0** and **EV1**.

cardPresso has several wizards available to help you to build your script.

To access to Wizards menu, please click in the **Encode View » Wizards** tab.



STMicroelectronics

SRTAG-D

Property	Returns:	Method:
UID	*Bool*	isValid()
UID	*String*	read()
Security	*Bool*	isValid()
Security	*Bool*	login (*String* **keyType**, *String* **keyData**);
Security	*Bool*	loginRead (*String* **keyData**);
Security	*Bool*	loginWrite(*String* **keyData**);
Security	*Bool*	changeReadKey(*String* **oldReadKey**, *String* **newReadKey**);
Security	*Bool*	changeWriteKey (*String* **oldWriteKey**, *String* **newWriteKey**);

Data	*Bool*	isValid()
Data	*Int*	size()
Data	*Bool*	write(*Int* **fileId**, *Int* **offset**, *String* **data**)
Data	*QScriptValue*	read(*Int* **fileId**, *Int* **offset**, *Int* **maxLength**)
Data (NFC)	*QScriptValue*	formatNfc(*String* **writeKey**, *String* **readKey**)
Data (NFC)	*QScriptValue*	writeNfcText(*String* **writeKey**, *String* **text**, *String* **locale**, *String* **encoding**, *String* **readKey**)
Data (NFC)	*QScriptValue*	writeNfcUri(*String* **writeKey**, *String* **uri**, *String* **readKey**)
Data (NFC)	*QScriptValue*	writeNfcIcon(*String* **writeKey**, *String* **icon**, *String* **readKey**)
Data (NFC)	*QScriptValue*	writeNfcSmartPoster (*String* **writeKey**, *String* **title**, *String* **uri**, *String* **action**, *String* **locale**, *String* **encoding**, *String* **readKey**)
Data (NFC)	*Bool*	prepareNfcVCard ()
Data (NFC)	*Bool*	addNfcVCardField(*String* **field**, *String* **data**)
Data (NFC)	*Bool*	writeNfcVCard(*String* **writeKey**, *String* **readKey**)
Data (NFC)	*Bool*	lockNfcApplication(*String* **writeKey**, *String* **readKey**)
Data (NFC)	*QScriptValue*	lockNfcApplicationForever(*String* **writeKey**, *String* **readKey**)
Data (NFC)	*QScriptValue*	unlockNfcApplication(*String* **writeKey**, *String* **readKey**)
Config	*Bool*	isValid()

UID

Bool **isValid()**

Returns "true" if the current UID property is implemented is valid and returns "false" if the current UID property is not implemented.

String **read()**

Required params: none.

Returns the Tag UID.

Security

✓

Bool **isValid()**

Returns "true" if the current Tag is successfully authenticated and returns "false" if the current is not authenticated.

Bool **login(**String **keyType,**String **keyData)**

Required params: "key", "data".

Authenticates the current Tag with the provided KeyType and a KeyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **loginRead**(*String* **keyData**)

Required params: "keyData".

Authenticates the current Tag with read only permissions with the provided KeyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **loginWrite**(*String* **keyData**)

Required params: keyData.

Authenticates the current Tag with read and write permissions with the provided KeyData. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **loginWrite**(*String* **oldReadKey**, *String* **newReadKey**)

Required params: oldReadKey, newReadKey.

Changes the current Read Key with the provided **newReadKey**. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Bool **changeWriteKey**(*String* **oldReadKey**, *String* **newReadKey**)

Required params: oldReadKey, newReadKey.

Changes the current Write Key with the provided **newReadKey**. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

Data

Bool **isValid()**

Returns "true" if the current Data property is implemented is valid and returns "false" if the current Data property is not implemented.

Int **size()**

Required params: none.

Retrives the total size of the current Tag as an integer value.

Bool **write**(*Int* **fileId**, *Int* **offset**, *String* **data**)

Required params: fileId, offset, data.

Writes the **data** defined by the user, in the file specified by the **fileId** and the specified **offset** in the Tag. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

QScriptData **read**(*Int* **fileId**, *Int* **offset**, *Int* **maxLength**)

Required params: fileId, offset, maxLength.

Retrieves the data from the file specified by the **fileId** in the specified **offset** and with the specified **maxLength**. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successfull or an error value if there was an error during the read process.

Bool **formatNfc**(*String* **writeKey**, *String* **readKey**)

Required params: writeKey, readKey.

Formats the Tag to NFC data.
writeKey refers to the key that allows writing in the tag.
readKey refers to the key that allows reading the tag

QScriptValue **writeNfcText**(*String* **writeKey**, *String* **text**, *String* **locale**, *String* **encoding**, *String* **readKey**)

Required params: writeKey, text, locale, encoding, readKey.

Writes a plain text object with the user defined **text** to an NFC formatted tag.

The **locale** parameter defines the language being used.
If no **locale** is provided the default locale will be "en-US"

The **encoding** parameter defines the character set being used.
If no **encoding** is provided the default locale will be "UTF8"
Supported charsets are **UTF8** and **UTF16**.

QScriptValue **writeNfcUri**(*String* **writeKey**, *String* **uri**, *String* **readKey**

Required params: writeKey, uri, readKey.

Writes a URI object with the user defined **uri** to an NFC formatted tag.

QScriptValue **writeNfcIcon**(*String* **writeKey**, *String* **icon**, *String* **readKey**)

Required params: writeKey, icon, readKey.

Writes an **icon** object (image) to an NFC formatted tag.

QScriptValue **writeNfcSmartPoster**(*String* **writeKey**, *String* **title**, *String* **uri**, *String* **action**, *String* **locale**, *String* **encoding**, *String* **readKey**)

Required params: appKey, icon.

Writes a smart poster object with the user defined **title** and **uri** to an NFC formatted tag.

The **action** parameter suggests a course of action that the device (reader) should do with the content.

The following actions are supported:

DO: runs/opens the provided data (send an sms / open a browser / perform a telephone call).
SAVE: saves the provided data (store an sms / save the URI in a bookmark / save the telephone number in contacts).
EDIT: opens the provided data for editing (open an sms in an sms editor / open an URI in a URI editor / open the telephone number for editing).

Note: The action will only be executed if the device (reader) supports the provided data requested action.

Note:
The **locale** parameter defines the language being used.
If no **locale** is provided the default locale will be "en-US"

The **encoding** parameter defines the character set being used.
If no **encoding** is provided the default locale will be "UTF8"
Supported charsets are **UTF8** and **UTF16**.

QScriptValue **prepareNfcVCard**()

Required params: none.

Creates a new vCard record with several commonly used vCard Fields ready for usage.

New vCard Fields can be added with the addNfcVCardField method.

After editing, the writeNfcVCard command must be used, to save the vCard data in the tag.

The default VCard fields created by the **prepareNfcVCard** method are:

first_name	middle_name	family_name	prefix_name	suffix_name	nickname	email
email_alternative	pager	fax	home_phone	cell_phone	business_phone	pro_Address1
pro_Address2	pro_Post_Code	pro_Region_State	pro_Town	pro_Country	address1	address2
post_Code	region_State	town	country	birthday	photo	role
title	company	note				

QScriptValue **addNfcVCardField**(*String* **field**, *String* **data**)

Required params: field, data.

Adds a new field to an existing NFC vCard (see prepareNfcVCard).
The field is identified by the **field** parameter and will contain the data provided in the **data** parameter.

QScriptValue **writeNfcVCard**(*String* **writeKey**, *String* **readKey**)

Required params: nfcKey.

Saves the data from an existing NFC vCard object (see prepareNfcVCard) in the tag.

QScriptValue **lockNfcApplication**(*String* **writeKey**, *String* **readKey**)

Required params: nfcKey.

Locks the NFC Application, preventing further changes to the saved data until the application is unlocked with the unlockNfcApplication command.

QScriptValue **lockNfcApplication**(*String* **writeKey**, *String* **readKey**)

Required params: nfcKey.

Permanently locks the NFC Application, preventing further changes to the saved data.

Bool **lockNfcApplication**(*String* **writeKey**, *String* **readKey**)

Required params: nfcKey.

Unlocks a previosuly locked NFC Application, allowing further changes to the saved data.

Config

Bool **isValid()**

Returns "true" if the current Config property is implemented is valid and returns "false" if the current Data property is not implemented.

SRTAG-D API

Bool **Verify**(*String* **passId**, *String* **keyData**)

Required params: passId, keyData.

Verifies if NFC APP EXISTS?

Bool **ChangeReferenceData**(*String* **passId**, *String* **keyData**)

Required params: passId, keyData.

Changes the reference data to the user provided **keyData**.

Bool **DisableVerificationRequirement**(*String* **passId**)

Required params: passId.

Disables the verification requirements of the tag.

Bool **SelectApplication**(*String* **appId**)

Required params: appId.

Selects the specified application.

Bool **SelectFile**(*String* **fileId**)

Required params: fileId.

Selects the specified file within the currently selected application.

Bool **CapabilityContainerSelect**()

Required params: none.

Selects the NFC Capability Container file.

Bool **NDEFSelect**()

Required params: none.

Selects the NDEF message file.

Bool **SystemFileSelect**()

Required params: none.

Selects the NFC system file.

Bool **enablePermanentState**(*String* **securityFlag**)

Required params: fileId.

Enables permanent state for the current tag.

Contactless APIs

NFC

The NFC Tag API provides methods to perform advanced NFC data handling, specifically to manipulate NDEFMessage objects, where several NFC record objects (text, uri, icon, smart poster, vcard) can be saved. This allows for a more customizable data structure development when working with NFC formatted data.

Property	Returns:	Method:
apis.nfc	*QScriptValue*	formatNfc(String masterKey,String **nfcKey**)
apis.nfc	*QScriptValue*	lockNfcApplication(String nfcKey)
apis.nfc	NDEFMessage	createNDEFMessage()
apis.nfc	*QScriptValue*	readNDEFMessage(String **appKey**);
apis.nfc	*QScriptValue*	writeNDEFMessage(String **appKey**,cpNDEFMessage **ndefMsg**);
apis.nfc	*NDEFTextRecord*	createNfcTextRecord(String **nfcKey**, String **text**, String **locale**, String **encoding**)
apis.nfc	NDEFUriRecord	createNfcUriRecord(String **nfcKey**, String **uri**)
apis.nfc	NDEFIconRecord	createNfcIconRecord(String **nfcKey**, QScriptValue **icon**)
apis.nfc	cpNDEFSmartPosterRecord	createNfcSmartPosterRecord(String **nfcKey**, String **title**, String **uri**, String **action**, String **locale**, String **encoding**)
apis.nfc	cpNDEFvCardRecord	createNfcvCardRecord()

formatNfc

QScriptValue **formatNfc(**String **masterKey,**String **nfcKey)**

Required params: masterKey.

Configures the DESFire card to receive NFC data.

Note: This command creates an NFC application where the NFC data will be saved.
masterKey referes to the PICC master key.
nfckey referes to the key of the NFC application
If no **nfcKey** is provided the PICC master key will be used.

lockNfcApplication

QScriptValue **lockNfcApplication**(*String* **nfcKey**)

Required params: nfcKey.

Locks the NFC Application, preventing further changes to the saved data.

createNDEFMessage

NDEFMessage **createNDEFMessage**()

Required params: none.

Creates a new NDEFMessage object where NFC records can be added.

The NDEFMessage object contains an **addRecord** command which can be used to store existing NFC records in the NDEFMessage object.

Void **addRecord**(nfcRecord **recordOjbect**)

readNDEFMessage

QScriptValue **readNDEFMessage**(*String* **appKey**);

Required params: none.

Reads the NDEFMessage currenly saved in the NFC application.

writeNDEFMessage

QScriptValue **writeNDEFMessage**(*String* **appKey**,*cpNDEFMessage* **ndefMsg**);

Required params: none.

Writes the specified **ndefMsg** object in the NFC application.

createNfcTextRecord

NDEFTextRecord **createNfcTextRecord**(*String* **nfcKey**, *String* **text**, *String* **locale**, *String* **encoding**)

Required params: appKey, text.

Creates a plain text NFC record object with the user defined **text**.

This record file can be added to an existing NDEFMessage object using the addRecord method.

The **locale** parameter defines the language being used.
If no **locale** is provided the default locale will be "en-US"

The **encoding** parameter defines the character set being used.
If no **encoding** is provided the default locale will be "UTF8"
Supported charsets are **UTF8** and **UTF16**.

createNfcUriRecord

NDEFUriRecord **createNfcUriRecord**(*String* **nfcKey**, *String* **uri**)

Required params: appKey, uri.

Creates a URI record object with the user defined **uri**.

This record file can be added to an existing NDEFMessage object using the addRecord method.

createNfcIconRecord

NDEFIconRecord **createNfcIconRecord**(*String* **nfcKey**, *QScriptValue* **icon**)

Required params: appKey, icon.

Creates an **icon** record object (image)..

This record file can be added to an existing NDEFMessage object using the addRecord method.

createNfcSmartPosterRecord

cpNDEFSmartPosterRecord **createNfcSmartPosterRecord**(*String* **nfcKey**, *String* **title**, *String* **uri**, *String* **action**, *String* **locale**, *String* **encoding**)

Required params: appKey, icon.

Creates a smart poster record object with the user defined **title** and **uri**.

This record file can be added to an existing NDEFMessage object using the addRecord method.

The **action** parameter suggests a course of action that the device (reader) should do with the content.

The following actions are supported:

DO: runs/opens the provided data (send an sms / open a browser / perform a telephone call).
SAVE: saves the provided data (store an sms / save the URI in a bookmark / save the telephone number in contacts).
EDIT: opens the provided data for editing (open an sms in an sms editor / open an URI in a URI editor / open the telephone number for editing).

Note: The action will only be executed if the device (reader) supports the provided data requested action.

Note:
The **locale** parameter defines the language being used.
If no **locale** is provided the default locale will be "en-US"

The **encoding** parameter defines the character set being used.
If no **encoding** is provided the default locale will be "UTF8"
Supported charsets are **UTF8** and **UTF16**.

createNfcvCardRecord

cpNDEFvCardRecord **createNfcvCardRecord**()

Required params: none.

Creates a new VCard record with several commonly used VCard Fields.

This record file can be added to an existing NDEFMessage object using the addRecord method.

New VCard Fields can be added with the addNfcVCardField method.

After editing, the writeNfcVCard command must be used, to save the VCard data in the tag.

The default VCard fields created by the **createNfcvCardRecord** method are:

first_name	middle_name	family_name	prefix_name	suffix_name	nickname	email
email_alternative	pager	fax	home_phone	cell_phone	business_phone	pro_Address1
pro_Address2	pro_Post_Code	pro_Region_State	pro_Town	pro_Country	address1	address2
post_Code	region_State	town	country	birthday	photo	role
title	company	note				

NFC Record Objects

NFC Record Objects provide access to methods that allow advanced operations with NFC Objects.

The Following NFC Objects are available:

Object:	
NDEFTextRecord	Holds and Handles NFC Text Record data
NDEFUriRecord	Holds and Handles NFC Uri Record data
NDEFIconRecord	Holds and Handles NFC Icon Record data
NDEFSmartPosterRec	Holds and Handles NFC Smart Poster Record data

ord

NDEFvCardRecord	Holds and Handles NFC Card Record data

NDEFTextRecord objects can be added to NDEFMessage objects (createNDEFMessage). They provide access to the methods which allow the user to store and manipulate plain text data in the record.

NDEFTextRecord objects support every Common Record Methods and the following methods:

Returns:	Method:
String	**getText**()
Void	**setText**(*String* **text**)
String	**getEncoding**()
Void	**setEncoding**(*String* **encoding**)
Int	**getLocale**()
Void	**setLocale**(*String* **locale**)

ScriptValue **getLocale**()
Required params: none.

Returns the locale of the Text record object.

Bool **setLocale**(String **locale**)
Required params: locale.

Returns the Type Name Format of the text record object.

ScriptValue **getText**()
Required params: none.

Returns the text of the Text record object.

Bool **setText**(String **text**)
Required params: text.

Sets the text value of the text record object.

ScriptValue **getEncoding**()
Required params: none.

Returns the encoding of the Text record object.

Bool **setEncoding**(String **encoding**)
Required params: encoding.

Sets the encoding of the text record object.

NDEFUriRecord objects can be added to NDEFMessage objects (createNDEFMessage). They provide access to the methods which allow the user to store and manipulate Uri data in the record.

NDEFUriRecord objects support every Common Record Methods and the following methods:

Returns:	Method:
String	getUri()

Void	setUri(*String* **Uri**)

ScriptValue **getUri**()

Required params: none.

Returns the Uri value of the Uri record object.

Bool **setUri**(String **uri**)

Required params: uri.

Sets the Uri value of the Uri record object.

NDEFIconRecord objects can be added to NDEFMessage objects (createNDEFMessage). They provide access to the methods which allow the user to store and manipulate Icon data in the record.

NDEFIconRecord objects support every Common Record Methods and the following methods:

Returns:	Method:
String	getData()
Void	setData(*String* **data**)

ScriptValue **getData**()

Required params: none.

Returns the icon data stored in the icon record object.

Bool **setData**(String **data**)

Required params: data.

Writes the icon data in the icon record object.

NDEFSmartPosterRecord objects are composed of several NFC Record Objects. After edited, NDEFSMartPosterRecord objects can be added to NDEFMessage objects (createNDEFMessage). They provide access to the methods which allow the user to store and manipulate Smart Poster data in the record.

NDEFSmartPosterRecord objects support every Common Record Methods and the following methods:

Returns:	Method:
String	hasTitle(*String* **locale**)
Void	getTitleCount()
NDEFTextRecord	getTitleRecord(*Int* **index**)
NDEFTextRecord list	getTitleRecords()
String	getTitle(*String* **locale**)
Bool	addTitle(*String* **text**, *String* **locale**, *String* **encoding**)
Bool	removeTitle(*String* **indexOrLocale**)
NDEFUriRecord	getUriRecord()
String	getUri()
Bool	setUri(*string* **Url**)
Bool	hasAction()
String	getAction()

Bool	setAction(*string* **action**)
Bool	hasSize()
Int	getSize()
Bool	setSize(*Int* **size**)
Bool	hasTypeInfo()
String	getTypeInfo()
Bool	setTypeInfo(*Int* **type**)
Bool	hasIcon(*String* **mimeType**)
Int	getIconCount()
NDEFIconRecord	getIconRecord(*Int* **index**)
NDEFIconRecord list	getIconRecords()
	getIcon(*String* **mimeType**)
	addIcon(*String* **type**, *QScriptValue* **iconData**)
	removeIcon(*String* **iconOrType**)

Bool **hasTitle**(*String* **locale**)

Required params: none.

Returns if the Smart Poster Record has a title. A locale value can be specified to detect if a specific locale has a designated title.

Bool **getTitleCount**()

Required params: none.

Returns the ammount of title records the Smart Poster Record has.

ScriptValue **getTitleRecord**(*Int* **index**)

Required params: none.

Returns an NDEFTextRecord saved in the NDEFSmartPosterRecord from the specified **index** in the Smart Poster record object.
If no **index** is specified 0 (zero) will be used as the default value.

ScriptValue **getTitleRecords**()

Required params: none.

Returns an indexed list with all the title records in the Smart Poster record object.

String **getTitle**(*String* **locale**)

Required params: none.

Returns if the Smart Poster Record has a title. A locale value can be specified to detect if a specific locale has a designated title.

Bool **addTitle**(*String* **text**, *String* **locale**, *String* **encoding**)

Required params: text.

Adds a Title to the Smart Poster record.

The **locale** parameter defines the language being used.
If no **locale** is provided the default locale will be "en-US"

The **encoding** parameter defines the character set being used.
If no **encoding** is provided the default locale will be "UTF8"

Supported charsets are **UTF8** and **UTF16**.

Note: There may be several Title records in a single Smart Poster record. However there can only be one Title record for each language identifier (locale).

Bool **removeTitle**(*String* **indexOrLocale**)
Required params: none.

Removes the title from the Smart Poster record. Titles to be removed can be specified by either the index or the locale.

ScriptValue **getUriRecord**()
Required params: none.

Returns the NDEFUriRecord saved in the NDEFSmartPosterRecord.

String **getUri**()
Required params: none.

Returns if the current Uri saved in the Uri record.

String **setUri**(*string* **Url**)
Required params: Url.

Sets the Uri in the uriRecord object.

Bool **hasAction**()
Required params: none.

Returns if the Smart Poster record has an action set.

String **getAction**()
Required params: none.

Returns the action currently configured in he Smart Poster record.

String **setAction**(*string* **action**)
Required params: Url.

Sets the action in the Smart Poster object.

The **action** parameter suggests a course of action that the device (reader) should do with the content.

The following actions are supported:

DO: runs/opens the provided data (send an sms / open a browser / perform a telephone call).
SAVE: saves the provided data (store an sms / save the URI in a bookmark / save the telephone number in contacts).
EDIT: opens the provided data for editing (open an sms in an sms editor / open an URI in a URI editor / open the telephone number for editing).

Bool **hasSize**()
Required params: none.

Returns if the Smart Poster record has a size record set.

String **getSize**()
Required params: none.

Returns the size currently configured in the size record of the Smart Poster record.

String **setSize**(*Int* **size**)
Required params: Url.

Sets the size record of the Smart Poster object with the user provided size value.

Bool **hasTypeInfo**()

Required params: none.

Returns if the Smart Poster record has a Type record set.

String **getTypeInfo**()

Required params: none.

Returns the type currently configured in the size record of the Smart Poster record.

String **setTypeInfo**(*Int* **type**)

Required params: Url.

Sets the type record of the Smart Poster object with the user provided type value.

The type value describes the type of object that can be reached trough the URI

Bool **hasIcon**(*String* **mimeType**)

Required params: none.

Returns if the Smart Poster record has any icon records set.

A specific MIME type can be specified when using this method.

Int **getIconCount**()

Required params: none.

Returns if the ammount of icon records currently saved in the Smart Poster record.

String **getIconRecord**(*Int* **index**)

Required params: none.

Returns the icon currently configured in the Smart Poster record.

ScriptValue **getIconRecords**()

Required params: none.

Returns an indexed list with all the icon records in the Smart Poster record object.

String **getIcon**(*String* **mimeType**)

Required params: none.

Returns the icon currently configured in the icon record of the Smart Poster record.
A specific MIME type can be specified when using this method.

QScriptValue **addIcon**(*String* **type**, *QScriptValue* **iconData**)

Required params: appKey, icon.

Adds an **icon** record (image) to the Smart Poster record.

Bool **removeIcon**(*String* **iconOrType**)

Required params: none.

Removes the icon from the Smart Poster record. Icons to be removed can be specified by either the icon index or the MIME type.

NDEFvCardRecord objects can be added to NDEFMessage objects (createNDEFMessage). They provide access to the methods which allow the user to store and manipulate vCard data in the record.

NDEFvCardRecord objects support every Common Record Methods and the following methods:

Returns:	Method:
String	addField(*String* **name**, *String* **value**)

QScriptValue **addField**(*String* **name**, *String* **value**)

Required params: name, value.

Adds a new field to an existing vCard record.
The field is identified by the **name** parameter and will contain the data provided in the **value** parameter.

NDEF record objects follow the NDEF specification. The following common methods are available for all NDEF record objects:

Returns:	Method:
String	getTypeNameFormat()
Void	setTypeNameFormat(*String* **TypeNameFormat**)
String	getType()
Void	setType(*String* **type**)
Int	getId()
Void	setId(*String* **Id**)
ScriptData	getPayload()
Void	setPayload(*String* **payLoad**)
ScriptValue	isEmpty()

ScriptValue **getTypeNameFormat**();

Required params: none.

Returns the Type Name Format of the record object.

Void **setTypeNameFormat**(String **typeNameFormat**);

Required params: typeNameFormat.

Sets the Type Name Format of the record object.

ScriptData **getType**();

Required params: none.

Returns the Type of the record object.

Void **setType**(String **type**);

Required params: type.

Sets the Type of the record object.

ScriptData **getId**();

Required params: none.

Returns the ID of the record object.

Void **setId**(String **ID**);

Required params: ID.

Sets the ID of the record object.

ScriptData **getPayload**();

Required params: none.

Returns the Payload of the record object.

Void **setPayload**(String **payload**);

Required params: payload.

Sets the payload of the record object.

Bool **isEmpty**();

Required params: none.

Returns if the record object is empty.

HID

ISO Prox

Enter topic text here.

UID

Bool **isValid()**

Returns "true" if the tag UID property is implemented for this tag type and "false" if not implemented.

String **read()**

Required params: none.

Returns the Tag UID.

Note:
If the Tag type does not support UID features the returned value will be undefined.

security

Bool **isValid()**

Returns "true" if the tag Security property is implemented for this tag type and "false" if not implemented.

Data

Bool **isValid()**

Returns "true" if the tag Data property is implemented for this tag type and "false" if not implemented.

Int **size()**

Required params: none.

Retrives the total size of the current Tag's memory as an integer value.

QScriptValue **read**(*Int* **offset**,*Int* **size**)

Required params: offset, size.

Retrieves the data from the Tag in the specified **offset** and with the specified **size**. Returns a QScriptValue object which inherits the respective properties and methods.

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successful or an error value if there was an error during the read process.

QScriptValue **write**(*Int* **offset**,*QScriptValue* **data**)

Required params: offset, data.

Writes the **data** defined by the user, in the specified **offset** in the Tag as a QScriptValue object. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

Config

Bool **isValid()**

Returns "true" if the tag Config property is implemented for this tag type and "false" if not implemented.

Int **size()**

Required params: none.

Retrives the total size of the current Tag configuration memory as an integer value.

4.4.5.Contact Tag Types

SLE4442

Property	Returns:	Method:
UID	*QScriptValue*	read ()
UID	*Bool*	isValid ()
Security	*QScriptValue*	login (*Byte* **PSC1**,Byte **PSC2**,Byte **PSC3**)
Security	*QScriptValue*	verifyPin(*Byte* **PSC1**,Byte **PSC2**,Byte **PSC3**)
Security	*QScriptValue*	changePin(*Byte* **newPSC1**,*Byte* **newPSC2**,*Byte* **newPSC3**)
Security	*QScriptValue*	login(*DataBlock* **Pin**)
Security	*QScriptValue*	verifyPin(*DataBlock* **Pin**)
Security	*QScriptValue*	changePin(*DataBlock* **Pin**)
Security	*QScriptValue*	readSecurityMemory()
Security	*QScriptValue*	updateSecurityMemory(*DataBlock* data)
Security	*QScriptValue*	updateSecurityMemory(*Int* **offset**,*DataBlock* **data**)
Data	*Int*	size()
Data	*QScriptValue*	read(*Int* **offset**,*int* **size**)
Data	*QScriptValue*	write(*Int* **offset**,*QScriptValue* **data**)
Config	*Int*	protectionMemorySize()
Config	*Int*	securityMemorySize()
Config	*QScriptValue*	readProtectionMemory()
Config	*QScriptValue*	writeProtectionMemory(*QScriptValue* **data**)

Config	*QScriptValue*	writeProtectionMemory(*Int* **offset**,*QScriptValue* **data**)

UID

read

String **read()**

Required params: none.

Returns the Tag UID as as string.

Note:
If the Tag type does not support UID features the returned value will be undefined.

isValid

Bool **isValid()**

Returns "true" if the current UID value is valid and returns "false" if the current UID value is false.

security

login

QScriptValue **login(***Byte* **PSC1,***Byte* **PSC2,***Byte* **PSC3)**

Required params: PSC1, PSC2, PSC3

Authenticates the current Tag Pins with three separate Byte values (**PSC1**,**PSC2**,**PSC3**) in either Integer or Hexadecimal format. Returns (Bool) 'True' if successful or 'False' if unsuccessful.

verifyPin

QScriptValue **.security.verifyPin(***Byte* **PSC1,***Byte* **PSC2,***Byte* **PSC3)**

Required params: PSC1, PSC2, PSC3

Validates the current Tag authentication Pins using three separate Bytes values (**PSC1**,**PSC2**,**PSC3**) in either Integer or Hexadecimal format. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

changePin

QScriptValue **changePin(***Byte* **newPSC1,***Byte* **newPSC2,***Byte* **newPSC3)**

Required params: newPSC1, newPSC2, newPSC3

Changes the current Tag authentication Pins using three separate Bytes values (**newPSC1,newPSC2,newPSC3**) in either Integer or Hexadecimal format. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

login

QScriptValue **login(***QScriptValue* **Pin)**

Required params: Pin

Authenticates the current Tag with a user specified **Pin**. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

verifyPin

QScriptValue **verifyPin(***QScriptValue* **Pin)**

Required params: Pin

Validates the current Tag authentication Pin with a user specified **Pin**. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

changePin

QScriptValue **changePin(***QScriptValue* **Pin)**

Required params: Pin

Changes the current Tag authentication Pin with a user specified **Pin**. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

readSecurityMemory

QScriptValue **readSecurityMemory**()

Required params: none

Retrieves the data from the Tag Security memory. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successful or an error value if there was an error during the read process.

updateSecurityMemory

QScriptValue **updateSecurityMemory**(*QScriptValue* **data**)

Required params: data

Writes **data** defined by the user in the the Tag Security memory. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

updateSecurityMemory

QScriptValue **updateSecurityMemory**(*Int*, **offset**,*QScriptValue* **data**)

Required params: offset, data

Writes the **data** defined by the user, in the specified **offset** in the Tag as a QScriptValue Object. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

Data

size

Int **size()**

Required params: none.

Retrives the total size of the current Tag as an integer value.

read

QScriptValue **read**(*Int* **offset**,*Int* **size**)

Required params: offset, size.

Retrieves the data from the Tag in the specified **offset** and with the specified **size**. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successful or an error value

if there was an error during the read process.

write

QScriptValue **write**(*Int* **offset**,*QScriptValue* **data**)

Required params: offset, data.

Writes the **data** defined by the user, in the specified **offset** in the Tag as a QScriptValue. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

Config

protectionMemorySize

Int **protectionMemorySize**()

Required params: none.

Retrieves the total size of the Tag Protection memory.

securityMemorySize

Int **securityMemorySize**()

Required params: none.

Retrieves the total size of the Tag Security memory.

readProtectionMemory

QScriptValue **readProtectionMemory**()

Required params: none.

Retrieves the data from the Tag Protection memory. Returns a QScriptValue Object which inherits the respective properties and methods.

Note:
Read methods contain an **.error** property which will return 'undefined' if the read was successful or an error value if there was an error during the read process.

writeProtectionMemory

QScriptValue **writeProtectionMemory**(*Int* **offset**,*QScriptValue* **data**)

Required params: data.

Writes the **data** defined by the user, in the specified **offset** in the Tag Protection memory as a QScriptValue. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

WARNING: This method will compare the input bytes with the data of the assigned address in memory card. If they are both the same, the protected bit of the address
will be set to 0. If the protected bit is 0,the address will not be allowed to update any more and the protected bit will never be set to 1.

writeProtectionMemory

QScriptValue **writeProtectionMemory**(*QScriptValue* **data**)

Required params: data.

Writes the **data** defined by the user, in the Tag Protection memory as a QScriptValue Object. Returns (QScriptValue) 'True' if successful or 'False' if unsuccessful.

WARNING: This method will compare the input bytes with the data of the assigned address in memory card. If they are both the same, the protected bit of the address
will be set to 0. If the protected bit is 0,the address will not be allowed to update any more and the protected bit will never be set to 1.

SLE4442 Sample Script

```
//return a list of the currently available contact encoders
encoders = encoding.encoders("CONTACT");


// for each of  the detected encoders, run the following script:
for (var iEncoder in encoders) {


   ///return the current encoder and log it
   encoder = encoders[iEncoder];
   log("Detected Encoder : " + encoder.name); //log the Encoder name


   //detect the Tag available in the encoder and log it
   tag = encoder.waitForTag("SLE.4442");
   log("Detected Tag : " + tag.type); //log the Tag name

   //login the SLE4442 tag and verify if login was successful
   authenticated = tag.security.login(data.fromHex("FFFFFF"));
   if (authenticated = "True") {

      //IMPORTANT: Failing to authenticate a Tag several times will result in locking the Tag permanently,
rendering it useless.

      //write sample data in the Tag main memory, from the offset (bit) 32 which is no longer part of the protection
memory
      myWriteData = tag.data.write(32, "Some sample data");

      //read the data to confirm it it was correctly saved in the Tag and if we can read from the Tag.
      myReadData = tag.data.read(32, 16);

         }
}
```

CARDPRESSO

more than an application

EMAII

Intervir+ Programa Operacional de Valorização do Potencial Económico e Coesão Territorial da RAM

QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL

REGIÃO AUTÓNOMA DA MADEIRA

UNIÃO EUROPEIA FEDER

s@cardpresso.com

www.cardPresso.com