

MADA UniC10 Plugin 4.17.x

API Documentation

Version 1.17.1



Inhaltsverzeichnis

Inhaltsverzeichnis	III
1 Overview	5
2 Registration	7
3 Memory Management	8
4 Implementation Guidelines	9
4.1 Asynchronous coding	9
4.2 Synchronous coding (DLL Version >= 4.16.0)	10
5 Function description	11
5.1 General functions	13
5.1.1 GetDLLVersion	13
5.1.2 SetLanguage	13
5.1.3 FreeWideChar	13
5.2 Registration functions	15
5.2.1 CheckRegistration	15
5.2.2 PerformRegistration	15
5.2.3 GetCustomerCode	16
5.2.4 ClearRegistration	16
5.3 Reader connection	18
5.3.1 GetRegisteredReaderNames	18
5.3.2 IsPCSCReader	18
5.3.3 ConnectReader	18
5.3.4 CheckReaderConnection	20
5.3.5 GetConnectedComPort	20
5.3.6 GetReaderSNR	21
5.3.7 DisconnectReader	21
5.3.8 SearchCards	22
5.3.9 SearchCardsExtraInfo	23

5.4	Coding setup.....	25
5.4.1	PrepareCoding.....	25
5.4.2	GetLoadedCommissionTransType.....	29
5.4.3	AnalyzeCodingMask	29
5.5	Legic®.....	32
5.5.1	ReadLegicMaster	32
5.5.2	CheckLegicMasters.....	32
5.5.3	ReadLegicMasterMemory	33
5.5.4	ReadLegicAdvantStamp.....	34
5.6	Coding - asynchronous	36
5.6.1	Code.....	36
5.6.2	SleepAndCode	37
5.6.3	CodeWithCallback	38
5.6.4	AbortCoding.....	38
5.6.5	GetCodingState.....	39
5.6.6	GetLog	40
5.6.7	GetCombiLog	41
5.6.8	FreeLog	42
5.7	Coding – synchronous	43
5.7.1	SyncCode	43
5.7.2	SyncCodeExtraInfo.....	44
5.8	Reading.....	48
5.8.1	LoadReadDefinition	48
5.8.2	ReadCard	48
6	Version History	50

1 Overview

UniC10 Plugin is a lightweight DLL interface to the MADA UniC10 Universal Coding software's core coding functionality. It allows connecting to a MADA RFID reader on a serial port and coding of RFID transponders (in the following simply called "cards").

The coding definition of the cards as well as the actual card type (e.g. LEGIC Prime®, Mifare DESFire, etc.) is defined in so-called coding masks or commissions.

A commission file is created by MADA Marx Datentechnik GmbH and defines card-type specific data and application structure which will be applied to all cards coded with the commission.

In addition to the fix data and application structure each commission allows the usage of user-definable data fields which can be filled separately for each coded card. There are two types of these user-definable data fields:

- a) Card ID: A decimal number that can be used for uniquely identifying a card in a more human-readable form than the card's unique serial number (UID). A commission can define more than one card ID field but all fields are filled with the same decimal input value. It is however possible to individually define for each card ID field how the decimal card ID value is to be coded onto the card. The following conversions are available:
 - HEX: Decimal value is directly coded onto the card. Card ID 123 would be coded as 0x7B.
 - ASCII: Decimal value is coded in ASCII representation. Card ID 123 would be coded as 0x303132.
 - BCD: Decimal value is coded in BCD representation. Card ID 123 would be coded as 0x123.
- b) Variable Data: A variable data field is an arbitrary area of data that can be filled in a highly customizable way. Usually the data of variable data fields stays the same for a batch of coded cards but of course it is also possible to change the data for every card. Typical examples for variable data fields would be the production year or the currency of value cards in a cash application. A variable data field is addressable by a unique name and the data entered by the user is predefined per variable data field as one of the following values:
 - ASCII: The user may enter a string of ASCII characters (0x20-0xFF).
 - HEX: Valid input characters are all Hexadecimal numbers ("0-9", "a-f" and "A-F").
 - Decimal: User may enter a decimal number that is only limited by the size of the variable data field.
 - BCD: Same as HEX, but only BCD characters ("0-9") are allowed.

Since DLL version 4.6.0 it is possible to load more than one commission at once.

This allows the coding of combi cards (for example a card containing both a LEGIC Prime® and a Mifare DESFire chip) in a single step if the connected reader supports all chips. Before version 4.6.0 the client application had to handle the coding of combi cards.

Also introduced in version 4.6.0 is a basic read functionality. This function works similar to the encoding process in that a so called read definition file is created by MADA Marx Datentechnik. This file defines the position and format of data on a transponder that already contains an encoding. The read function then allows reading the data at the position defined in the read definition file, usually a card ID number.

2 Registration

In order to be able to load and encode commissions which were created with an own installation of UniC10 SQL (and not provided by MADA Marx Datentechnik GmbH), the UniC10 Plugin DLL has to be registered first. UniC10 Plugin offers three modes of registration:

- 1) Online Registration: Registration is performed with a supplied serial number over an internet connection to the MADA registration server. In case there is no internet connection available, the online registration can also be performed by requesting an activation key by phone. The MADA registration server is contacted through port 1979, so that port needs to be accessible in the local network settings. Online registration is only valid for the system it was first performed on.
- 2) Dongle: As long as a valid dongle is inserted, registration is considered to be successfully performed on the system.
- 3) License File: The registration is stored in a file named "license.lic" which is created by MADA. This file has to be copied to the same directory as the UniC10 Plugin DLL. A license file is always linked to a specific hardware ID of one (or more) RFID reader which is also supplied and branded by MADA.

A registered UniC10 Plugin always contains a hidden customer code and only accepts commissions that are created on a system which contains the same customer code.

If the coding file is created by MADA Marx Datentechnik GmbH, no registration is necessary. The example commissions provided with this API package for example are commissions which require no registration.

3 Memory Management

Several UniC10 Plugin functions return string data either as a return value or as PWCHAR parameters. The memory for these strings is allocated inside the UniC10 Plugin DLL, so it is not necessary to allocate any buffers before calling such a function. To free this memory again, the DLL function *FreeWideChar* should be called for every PWCHAR value once its data is not used anymore.

4 Implementation Guidelines

A typical implementation of the UniC10 Plugin API consists of the following steps:

1. [OPTIONAL] If the used coding file requires a registration (i.e. it was not provided by MADA Marx Datentechnik GmbH), call the *CheckRegistration* function at least once for each new installation of the application to ensure that the software is registered. If *CheckRegistration* returns false and no dongle registration is available for the product, call *PerformRegistration*. If the application only wants to offer a registration by dongle the call to *PerformRegistration* can be skipped – the registration is active automatically as long as the dongle is inserted.
2. [OPTIONAL] Retrieve the list of registered readers with the *GetRegisteredReaderNames* function and display it to the user in a combobox.
3. Try to establish a connection to a reader (specified by its name and comport which was either selected by the user or for example predefined in an INI-File) with the *ConnectReader* function. If only one reader is connected and connection speed is of no concern, pass the reader name "AUTOCONNECT" to the *ConnectReader* function to automatically detect a compatible reader when *PrepareCoding* is called.
4. [OPTIONAL] Once a connection to a reader has been established start a thread that periodically checks the reader connection every second with the *CheckReaderConnection* function. As soon as the function fails, notify the user that the reader connection has been lost and call the *Disconnect* function.
5. Initialize the coding by calling the *PrepareCoding* function. The function needs the path to the .xml coding mask file that describes the coding and the handle to the window processing the WM_CREATE_CARD callback messages (if asynchronous coding is used). If more than one chip should be encoded in a combi card, call the *PrepareCoding* function accordingly. [OPTIONAL] Use the *hasCardID*, *maxCardID* and *varData* values to display commission-specific input fields for the user.
6. [OPTIONAL] If the coded commission is a Legic® commission, check which master data has to be loaded to the reader by calling the *CheckLegicMasters* function. Repeatedly add masters by calling the *ReadLegicMasters* function until all necessary masters have been loaded to the reader.

At this point there are two possible approaches to encode the actual cards: Synchronous and asynchronous coding. The synchronous coding was added in DLL version 4.16.0 and offers the advantage of a simpler implementation. On the downside, the user has to handle the background threading of the coding on the client side. Since codings may take several seconds (depending on the coding mask, 10-30 seconds are possible) it is recommended to either perform the synchronous coding in a background thread or use the asynchronous approach provided by the UniC10 Plugin DLL.

4.1 Asynchronous coding

For each card that is to be programmed:

- a. Call the Code function with the appropriate parameters (*cardID*, *varData*) for the current card.

- b. If after a set amount of time no CODING_START message arrives, call the function *AbortCoding* and display an error. This can happen if a card is moved to the reader that contains no chip or a wrong chip type.
 - c. [OPTIONAL] Display a message “Searching for card” and once the CODING_START message arrives, display “Coding...”.
 - d. [OPTIONAL] For each incoming CODING_STEP message, increase the position of the progress bar that displays the coding progress.
 - e. [OPTIONAL] If a combi coding is performed, visualize that the coding of one chip is finished as soon as a CARD_FINISHED message arrives.
 - f. The coding of the card is finished once one of the messages CODING_SUCCESS, CODING_FAIL or CODING_ERROR arrives.
 - g. [OPTIONAL] Retrieve the log data for the card by calling the *GetLog* function with the log handle supplied in CODING_SUCCESS or CODING_FAIL. Store this log in a log file or in a database for future reference. If a combi coding was performed, use the *GetCombiLog* function instead.
 - h. Free the log data by calling the *FreeLog* function with the log handle supplied in CODING_SUCCESS or CODING_FAIL.
7. When the last card was coded, stop the coding by calling *AbortCoding*, disconnect the reader by calling the *DisconnectReader* function and unload the UniC10 Plugin DLL.

4.2 Synchronous coding (DLL Version >= 4.16.0)

For each card that has to be programmed:

- a. Call the *SyncCode* function with the appropriate parameters (*cardID*, *varData*) for the current card, ideally in a background thread which not blocks the UI Thread. The *SyncCode* function returns once the coding is completed.
- b. The return parameters of the *SyncCode* function contain the card’s UID as well as the outcome of the coding including error messages if applicable.
- c. Free the string return parameters by calling the *FreeLog* function.

5 Deployment

When deploying the application that uses UniC10 Plugin DLL, it has to be ensured that the file matrix32.dll is also deployed in the same folder as UniC10Plugin.dll.

6 Function description

This section describes the functions exported by the UniC10 Plugin DLL in detail.

The functions can be organised into the following sections:

- General – Functions related to the UniC10 Plugin DLL
- Registration – Registration related functions. Since registration is only necessary for self-created coding masks (and not coding masks provided by MADA Marx Datentechnik GmbH), this section can be ignored depending on the nature of the coding masks.
- Reader connection – Functions for setting up a connection to an RFID reader. A reader connection is necessary prior to setting up the encoding.
- Coding setup – Functions for preparing and loading the coding mask(s).
- Legic – Optional functions for handling master data.
- Coding – Functions for performing the card by card coding.
- Reading – Functions related to the optional card reading functionality.

6.1 General functions

This section describes general functions related to the UniC10 Plugin DLL itself.

6.1.1 GetDLLVersion

PWCHAR GetDLLVersion()

Returns a string describing the loaded DLL's version.

In order to avoid memory leaks, the return value should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In	
Out	
Return	
dllVersion	Version of the DLL in the form <Major>.<Minor>.<Release> r<Build>, for example "1.0.0 r1234".

6.1.2 SetLanguage

void SetLanguage(PWCHAR lang)

Sets the language of all dialogs and error messages used by UniC10 Plugin.

In	
lang	Language code of the language to use. Currently German "de" and English "en" are available.
Out	
Return	

6.1.3 FreeWideChar

void FreeWideChar(PWCHAR wideChar)

Frees the memory that was allocated for the string the PWCHAR parameter points to. To avoid memory leaks, this function should be called for every PWCHAR return value that was retrieved by a UniC10 Plugin function.

In	
wideChar	Pointer to the wide char string as returned by a UniC10 Plugin function, for example <i>GetRegisteredReaderNames</i> .
Out	
Return	

6.2 Registration functions

Registration related functions. Since registration is only necessary for self-created coding masks (and not coding masks provided by MADA Marx Datentechnik GmbH), this section can be ignored depending on the nature of the coding masks.

6.2.1 CheckRegistration

int CheckRegistration()

Checks if the UniC10 Plugin DLL is registered and contains a valid customer code.

In	
Out	
Return	
0	Registration valid and commissions with the registered customer code can be coded.
1	Registration invalid. <i>PrepareCoding</i> and <i>Code</i> functions will return with error code 4. To perform the registration procedure, call <i>PerformRegistration</i> . Note that for dongle registrations it is sufficient to simply insert the dongle, a call to <i>PerformRegistration</i> is not necessary. If registration is done with a license file, this error code is always returned.

6.2.2 PerformRegistration

int PerformRegistration(HWND parent)

Opens the online registration dialog. Please note that dongle registrations are performed automatically and do not require a call to this function.

In	
parent	Handle of the window that should be the parent of the dialogs that appear during the registration. The registration dialogs only appear if the UniC10 Plugin DLL is not currently registered. The registration procedure only needs to be performed once per installation.
Out	

Return	
0	Registration was completed successfully and commissions with the registered customer code can be coded.
1	Registration invalid. <i>PrepareCoding</i> and <i>Code</i> functions will return with error code 4. To perform the registration procedure, call <i>PerformRegistration</i> again.
2	Already registered. Performing the registration procedure is not necessary. To re-perform a registration (for example to change the customer code), first call the <i>ClearRegistration</i> function or, in case of a dongle registration, remove the dongle.

6.2.3 GetCustomerCode

PWCHAR GetCustomerCode()

Retrieves the customer code of the currently active registration.

Please note that if registration is done with a license file, a correctly branded reader (which matches the license file) needs to be connected with the function *ConnectReader* first, otherwise no customer code can be returned by this function.

In order to avoid memory leaks, the return value should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In	
Out	
Return	
customerCode	A hex string containing the customer code of the current registration or an empty string if the product is not registered.

6.2.4 ClearRegistration

void ClearRegistration()

Deletes the current registration. This function has to be called first if a registration (for example the customer code) should be changed. Afterwards the new registration can be performed again with the *PerformRegistration* function.

In
Out
Return

6.3 Reader connection

Functions for setting up a connection to an RFID reader. A reader connection is necessary prior to setting up the encoding.

6.3.1 GetRegisteredReaderNames

void GetRegisteredReaderNames(PWCHAR &readerNames)

Returns a list of valid reader names which can be used in the *ConnectReader* function.

In order to avoid memory leaks, the *readerNames* pointer should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In	
Out	
readerNames	A list separated by STX control characters (0x02) of reader names that can be used as parameter <i>readerName</i> in the <i>ConnectReader</i> function. The last reader name is also terminated by an STX character.
Return	

6.3.2 IsPCSCReader

int IsPCSCReader(PWCHAR readerName)

Checks if a specific reader is connected via PCSC protocol, like the Omnikey 5321 CL reader for example. This function can be used to decide if it is necessary to enter RS232 specific parameters like com port and baudrate on the user frontend. PCSC readers do not require these parameters.

In	
Out	
Return	
0	The specified reader is no PCSC reader but connected to an RS232 port.
1	The specified reader is a PCSC reader connected via USB port.

6.3.3 ConnectReader

int ConnectReader(PWCHAR readerName, BYTE comport, unsigned int baudrate, BYTE stopbits, BYTE databits, BYTE parity)

Attempts to connect to the reader specified by *readerName*. If a connection could be established, the coding functions (*PrepareCoding*, *Code*) can be used.

If a reader is already connected before this function is called, it is disconnected first.

In	
readerName	<p>The name of the reader to connect to. Valid and registered reader names should be retrieved with the function <i>GetRegisteredReaderNames</i>.</p> <p>If the string "AUTOCONNECT" is passed as reader name, UniC10 Plugin DLL enters autoconnect mode and the actual reader connection is performed automatically during a call to <i>PrepareCoding</i>. Autoconnect mode is disabled once <i>ConnectReader</i> is called with a name other than "AUTOCONNECT" or when <i>DisconnectReader</i> is called. If autoconnect mode should only be done for a certain reader type, specify the readerName as "AUTOCONNECT:<readerName>", for example "AUTOCONNECT:Legic Advant SM4500". This speeds up the autoconnect process a lot because not all possible reader types have to be checked.</p> <p>If the reader should be connected via TCP/IP (only specific MADA installations!), the IP address and TCP port is appended to the reader name, separated by an @ character, for example "Multiband@192.168.0.13:10001".</p>
comport	Serial comport the reader is connected to. If the connected reader is a PCSC reader or connection is done via TCP/IP, this value is ignored.
baudrate	Baudrate of the reader. If 0 is specified, the reader's default baudrate is used. If the connected reader is a PCSC reader, this value is ignored.
stopbits	<p>Stopbits of the reader. Valid values are:</p> <ul style="list-style-type: none"> • 0: 1 stop bit • 1: 1.5 stop bits • 2: 2 stop bits <p>If 0xFF is specified, the reader's default stop bit setting is used. If the connected reader is a PCSC reader, this value is ignored.</p>
databits	The reader's databits. If 0xFF is specified, the reader's default databits setting is used. If the connected reader is a PCSC reader, this value is ignored.
parity	<p>The reader's parity. Valid values are:</p> <ul style="list-style-type: none"> • 0: No parity • 1: Odd parity • 2: Even parity

	<ul style="list-style-type: none"> • 3: Mark parity • 4: Space parity <p>If 0xFF is specified, the reader's default parity is used. If the connected reader is a PCSC reader, this value is ignored.</p>
Out	
Return	
0	Connection was successful.
1	Reader name invalid.
2	Connection could not be established.

6.3.4 CheckReaderConnection

int CheckReaderConnection()

Checks if the current reader is still connected.

This function can be used to determine when a connected reader is no longer accessible (e.g. if the power supply is interrupted or the serial cable is disconnected) and can be called periodically to monitor the reader connection state.

When a lost connection is detected, the function *DisconnectReader* should be called.

If the connected reader is a "Legic Advant SM4500" reader, calling this function also resets the master memory timeout to the max value (255 minutes). There is no other way to refresh the master timeout for this specific reader, so this function should be called periodically to avoid having to reload Legic® masters.

In	
Out	
Return	
0	Reader connected.
1	Reader not connected.
2	No reader connected but DLL is in autoconnect mode.

6.3.5 GetConnectedComPort

int GetConnectedComPort(BYTE &comport)

Returns the comport the currently connected reader is connected to.

If the reader does not connect by comport, for example a PCSC reader, the comport parameter has the value 0.

This function can be used to check which comport a reader was connected to in autoconnect mode. This comport could then be used as a first try in future connection attempts to speed up the connection process.

In	
Out	
Return	
0	Reader connected.
1	Reader not connected.

6.3.6 GetReaderSNR

PWCHAR GetReaderSNR()

Retrieves the serial number of the currently connected reader.

In order to avoid memory leaks, the return value should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In	
Out	
Return	
readerSNR	A hex string containing the serial number of the currently connected reader or an empty string if the connected reader has no serial number or there is no reader connected.

6.3.7 DisconnectReader

void DisconnectReader()

Disconnects the currently connected reader and frees its comport for other applications or for re-connection.

This function should be called by the client application when a connection loss is detected by the *CheckReaderConnection* function in order to clean up the DLL's internal state.

If the DLL was currently In autoconnect mode, this mode is disabled after a call to this function.

In
Out
Return

6.3.8 SearchCards

int SearchCards(PWCHAR transType, PWCHAR &transDesc)

Reads information of every transponder that is currently in the connected reader's field. Note that this function is completely optional and is not necessary to use in a default process. Please refer to the Implementation Guidelines chapter for a description of the default process.

In order to avoid memory leaks, the *transDesc* pointer should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In	
transType	<p>The transponder type to search for. If an empty string is passed, the function returns all found transponders. If a transponder type is specified only the transponders that match this string are returned. This allows for faster selection because the reader does not have to search for every possible transponder type.</p> <p>To determine a certain chip's transponder type string, call this function with an empty <i>transType</i> parameter and check the <i>transponderType</i> field of the returned <i>transDesc</i> parameter.</p> <p>The transponder type string of the currently loaded commission can be obtained by calling the <i>GetLoadedCommissionTransType</i> function.</p>
Out	
transDesc	<p>A string containing descriptions of all found transponders. Each transponder description is represented by a string of the following format: <UID>[ETX]<transponderType>[ETX]<unknownDesc> where [ETX] is the control character 0x03.</p> <ul style="list-style-type: none"> • <i>UID</i> is a hexadecimal representation of the chip's UID (serial number), for example "3D002A55". If a chip has no UID (for example SLE4442), the UID is returned as "00". • <i>transponderType</i> uniquely identifies the chip type. For example a Legic® Prime chip will always have the transponder type "LegicPrime". If the transponder type is unknown or not supported by UniC10, the transponder type string is empty and a description of the chip can be found in the <i>unknownDesc</i> field.

	<ul style="list-style-type: none"> If a chip was found that is not supported by UniC10 and the corresponding transponder type is empty, a description of the chip type is found in the <i>unknownDesc</i> field. This description might change between releases so it should not be used to identify chip types. For example currently Mifare Plus chips are not supported by UniC10. For these chips the transponder type is empty and the unknownDesc is for example "Mifare Plus 4k SL2". <p>Multiple transponder descriptions (for multiple found transponders) are separated by an STX (0x02) character. The last string description is also terminated by an STX character.</p>
Return	
0	Execution successful.
1	No reader connected.
2	Invalid coding state. Has to be "Offline" or "Online" (see function <i>GetCodingState</i>).
3	Unknown transponder type.

6.3.9 SearchCardsExtraInfo

int SearchCardsExtraInfo(PWCHAR transType, PWCHAR &transDesc)

Reads information of every transponder that is currently in the connected reader's field. In addition to the *SearchCards* function, this function also reads additional chip information, e.g. "Mifare Desfire 4k EV1 v5" instead of just "Mifare Desfire". Since reading this additional information takes some time, execution time of this function is a bit slower than just calling *SearchCards*.

In order to avoid memory leaks, the *transDesc* pointer should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In	
transType	<p>The transponder type to search for. If an empty string is passed, the function returns all found transponders. If a transponder type is specified only the transponders that match this string are returned. This allows for faster selection because the reader does not have to search for every possible transponder type.</p> <p>To determine a certain chip's transponder type string, call this function with an empty <i>transType</i> parameter and check the <i>transponderType</i> field of the returned <i>transDesc</i> parameter.</p>

	The transponder type string of the currently loaded commission can be obtained by calling the <i>GetLoadedCommissionTransType</i> function.
Out	
transDesc	<p>A string containing descriptions of all found transponders. Each transponder description is represented by a string of the following format: <code><UID>[ETX]<transponderType>[ETX]<unknownDesc>[ETX]<chipType>[ETX]<chipVersion></code> where [ETX] is the control character 0x03.</p> <ul style="list-style-type: none"> • <i>UID</i> is a hexadecimal representation of the chip's UID (serial number), for example "3D002A55". If a chip has no UID (for example SLE4442), the UID is returned as "00". • <i>transponderType</i> uniquely identifies the chip type. For example a Legic® Prime chip will always have the transponder type "LegicPrime". If the transponder type is unknown or not supported by UniC10, the transponder type string is empty and a description of the chip can be found in the <i>unknownDesc</i> field. • If a chip was found that is not supported by UniC10 and the corresponding transponder type is empty, a description of the type is found in the <i>unknownDesc</i> field. This description might change between releases so it should not be used to identify chip types. For example currently Mifare Plus chips are not supported by UniC10. For these chips the transponder type is empty and the unknownDesc is for example "Mifare Plus 4k SL2". • <i>chipType</i> contains additional type information of the transponder, e.g. "MIM1024" or "4k EV1". • <i>chipVersion</i> contains additional version information of the transponder, e.g. "v4" or "v5" for Mifare Desfire chips or "NM" for Legic® Prime chips. <p>Multiple transponder descriptions (for multiple found transponders) are separated by an STX (0x02) character. The last string description is also terminated by an STX character.</p>
Return	
0	Execution successful.
1	No reader connected.
2	Invalid coding state. Has to be "Offline" or "Online" (see function <i>GetCodingState</i>).
3	Unknown transponder type.
4	Error reading chip type and version.

6.4 Coding setup

Functions for preparing and loading the coding mask(s). The only required function in this section is the function `PrepareCoding`, which has to be called before the actual coding can be started. All other functions in this section are optional and only should be used in special cases.

6.4.1 PrepareCoding

int PrepareCoding(PWCHAR commFileName, HWND codingCallback, HWND parent, int &estimatedCreationSteps, BYTE &hasCardID, PWCHAR &maxCardID, PWCHAR &varData)

Prior to the actual coding, this function has to be called once after a new commission has been loaded or a reader connection has been established.

If the DLL is in autoconnect mode, this function tries to automatically connect to a compatible reader if no appropriate reader connection has been established yet. Please note that depending on the number of installed COM Ports this function may take several seconds to complete if autoconnect mode is activated.

In order to avoid memory leaks, the *maxCardID* and *varData* pointers should be freed by passing them to the *FreeWideChar* function once they are not needed anymore.

In	
commFileName	<p>The absolute path of the xml file describing the coding. If more than one commission should be loaded in order to perform combi coding, there are two possibilities:</p> <ul style="list-style-type: none">• Specify more than one xml file in this parameter. Each file has to be separated by an STX (0x02) character.• Specify the path to a text file with the ending .cdf (Combi Definition File). This text file has to contain all file names (not the full path) of all XML coding files that should be loaded, each in a separate row. This option was added to allow combi coding without having to change the implementation of the UniC10 Plugin DLL. <p>Since Plugin version 4.17.0, this parameter can also contain the complete unparsed file content of the xml file instead of the file name. Multiple commissions also have to be separated by an STX (0x02) character. It is also possible to mix file names and xml content for multiple commissions.</p>
codingCallback	<p>Handle to the window that receives status messages of the coding process. The status message is defined as WM_CREATE_CARD (0x804D). The message's wParam defines the actual status:</p> <p>0 - CODING_START: A card has been found and the coding process is about to start.</p>

	<p>1 - CODING_SUCCESS: The card has been successfully coded. IParam contains the handle to logging information which can be obtained by the function <i>GetLog</i>.</p> <p>2 - CODING_FAIL: The coding could not be completed because of an error. IParam contains a handle to logging information which can be obtained by the function <i>GetLog</i>. The log can be stored by the client application or used for displaying the actual error that caused the coding to fail.</p> <p>3 - CARD_REMOVED: The coded card (either successful or failed) has been removed from the reader's field. This information can be used to determine when the next coding can start.</p> <p>4 - CODING_ERROR: An internal error has occurred and the coding of the card has been cancelled.</p> <p>5 - CODING_STEP: The coding has progressed one step. In conjunction with the <i>estimatedCreationSteps</i> parameter of the function <i>PrepareCoding</i>, this message can be used to determine the progress of the coding.</p> <p>6 - This value is never sent and reserved for future use.</p> <p>7 - CARD_FINISHED: In case of a combi coding, this message is sent every time one chip's coding process has finished. Note that for backwards compatibility reasons, this message is only sent if the combi coding was loaded with the first option of the <i>commFileName</i> parameter (multiple XML files separated by STX characters).</p> <p>Since version 4.8.0 of the DLL, it is possible to use an alternative to Windows messages as a means to receive the status messages of the coding process. To activate this alternative callback feature, the <i>codingCallback</i> parameter has to be set to 0. Also, instead of the <i>Code</i> function, the <i>CodeWithCallback</i> has to be used where a callback function can be specified which retrieves the wParam and IParam values according to the above specification. The callback mode can be useful in environments where Windows messages cannot be easily retrieved.</p>
parent	Handle of the window that should be the parent of possible pop-up windows that appear during the preparation phase. A pop-up

	<p>window appears for example if a LEGIC® prime commission is loaded and the specified reader is missing master cards mandatory for the coding of that commission.</p> <p>If this value is 0, no dialogs are opened and no warnings are displayed if any prerequisites for the encoding are not met. Instead, an appropriate error is thrown during card coding (for example NO_MASTER_CARD).</p>
Out	
estimateCreationSteps	<p>The number of steps the coding of a card with the loaded xml coding mask(s) will approximately need on the connected reader. This number can be used in conjunction with the CODING_STEP messages (see parameter <i>codingCallback</i>) to determine the current progress of the coding. If this number is 0, the number of steps needed to code the card is unknown.</p>
hasCardID	<p>Flag indicating whether at least one of the loaded commissions defines a card ID field. 0 means there is no commission with a card ID field, 1 means there is one (or more) card ID defined.</p>
maxCardID	<p>A string defining the maximum possible (decimal) card ID. The maximum card ID is limited by the size of the commissions' smallest card ID field. If the commissions define no cardID field, this parameter can be ignored. Note that the maximum cardID is not necessarily convertible into a 32-bit integer variable because there is no limit on the size of a cardID field.</p>
varData	<p>A string describing the loaded commissions' variable data fields. Each variable data field is represented by a string of the following format: <i><areaName></i>[ETX]<i><inputType></i>[ETX]<i><maxValue></i> where [ETX] is the control character 0x03. <i>areaName</i> is the name (and index) of the variable data field. <i>inputType</i> can be one of the following values, also defining the interpretation of the <i>maxValue</i> field:</p> <ul style="list-style-type: none"> • ASCII: Valid input values are all ASCII characters (0x20-0xFF). <i>maxValue</i> is to be interpreted as number of characters / bytes. If the entered input value is shorter than the maximum number of characters, the string is padded with 0x00 on the left side. • HEX: Valid input values are hexadecimal characters ("0-9", "a-f", "A-F"). <i>maxValue</i> is the maximum number of bits the variable

	<p>data field can hold. Example: If <i>maxValue</i> is 13, valid input values range from "0" to "1FFF".</p> <ul style="list-style-type: none"> • DEC: The input value is a decimal value (each character "0-9"). <i>maxValue</i> is the maximum decimal value that can be entered. • BCD: Valid input values are BCD characters ("0-9"). <i>maxValue</i> is the number of allowed BCD digits. <p>Multiple string descriptions (for multiple variable data fields) are separated by an STX (0x02) character. The last string description is also terminated by an STX character.</p>
Return	
0	Preparation successful. Coding State has been changed to "Online" (see function <i>GetCodingState()</i>).
1	File could not be loaded. Check access rights to the file name. If the file does not exist, the <i>commFileName</i> parameter is treated as XML content and will throw error code 8 since it is no valid XML file.
2	Reader not connected.
3	Reader cannot code the commission(s). This can happen for example if the connected reader is a Legic Prime MSM-S module and the commission specifies a Mifare DESFire coding.
4	License not valid. If a dongle license is used, it is sufficient to insert the dongle. Other licensing methods like the online registration however require a call to <i>CheckRegistration</i> at least once per installation.
5	CRC validation of one of the specified commission files failed. Either the file was manually changed or the file contains a different customer code.
6	Commission file deprecated. At least one of the commission files was created with a newer version of UniC10 and contains coding directives that are not known by the current version of UniC10 Plugin. Solve by updating the UniC10 Plugin DLL to the latest version.
7	Transponder type specified by one of the commission files is unknown because it was introduced by a later version of UniC10. Solve by updating the UniC10 Plugin DLL to the latest version.
8	Commission file(s) invalid. This error is also thrown, if a file name was passed to the <i>commFileName</i> parameter and the file could not be found.

9	At least one of the commissions cannot be used because it defines external data source fields. UniC10 Plugin currently does not support external data sources.
10	Tried to load a combi commission with a duplicate transponder type – for combi codings, all loaded commissions must be of a different transponder type. For example, it is not possible to load two Mifare DESFire commissions simultaneously.
11	Tried to autoconnect a reader but no reader was found that can encode all commissions.

6.4.2 GetLoadedCommissionTransType

int GetLoadedCommissionTransType(PWCHAR &transType)

Returns the transponder type string(s) of the loaded commission(s). This string can be used in the *SearchCards* function.

In order to avoid memory leaks, the *transType* pointer should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In	
Out	
transType	The loaded commission's transponder type string for use in the <i>SearchCards</i> function. If more than one commission was loaded for combi coding, the transponder types are separated by an STX (0x02) character. Note that the <i>SearchCards</i> function only allows passing one transponder type, so in case of a combi coding the <i>transType</i> string needs to be split up.
Return	
0	Execution successful.
1	No commission loaded.

6.4.3 AnalyzeCodingMask

int AnalyzeCodingMask(PWCHAR commFileName, PWCHAR &transType, BYTE &hasCardID, PWCHAR &maxCardID, PWCHAR &varData)

Gets the same information from a coding mask XML file that is also obtained by the *PrepareCoding* function – however for this function, it is not necessary to connect a reader.

In order to avoid memory leaks, the *maxCardID*, *transType* and *varData* pointers should be freed by passing them to the *FreeWideChar* function once they are not needed anymore.

In	
commFileName	The absolute path of the xml file.
Out	
transType	The specified commssion's transponder type.
hasCardID	Flag indicating whether the specified commission defines a card ID field. 0 means there is no card ID field, 1 means there is one.
maxCardID	A string defining the maximum possible (decimal) card ID. Note that the maximum cardID is not necessarily convertible into a 32-bit integer variable because there is no limit on the size of a cardID field.
varData	<p>A string describing the loaded commission's variable data fields. Each variable data field is represented by a string of the following format: <i><areaName></i>[ETX]<i><inputType></i>[ETX]<i><maxValue></i> where [ETX] is the control character 0x03. <i>areaName</i> is the name (and index) of the variable data field. <i>inputType</i> can be one of the following values, also defining the interpretation of the <i>maxValue</i> field:</p> <ul style="list-style-type: none"> • ASCII: Valid input values are all ASCII characters (0x20-0xFF). <i>maxValue</i> is to be interpreted as number of characters / bytes. If the entered input value is shorter than the maximum number of characters, the string is padded with 0x00 on the left side. • HEX: Valid input values are hexadecimal characters ("0-9", "a-f", "A-F"). <i>maxValue</i> is the maximum number of bits the variable data field can hold. Example: If <i>maxValue</i> is 13, valid input values range from "0" to "1FFF". • DEC: The input value is a decimal value (each character "0-9"). <i>maxValue</i> is the maximum decimal value that can be entered. • BCD: Valid input values are BCD characters ("0-9"). <i>maxValue</i> is the number of allowed BCD digits. <p>Multiple string descriptions (for multiple variable data fields) are separated by an STX (0x02) character. The last string description is also terminated by an STX character.</p>
Return	
0	XML file is valid and could be analysed.
1	File not found.
4	License not valid. If a dongle license is used, it is sufficient to insert the dongle. Other licensing methods like the online registration

	however require a call to <i>CheckRegistration</i> at least once per installation.
5	CRC validation of the specified commission file failed. Either the file was manually changed or the file contains a different customer code.
6	Commission file deprecated. The commission file was created with a newer version of UniC10 and contains coding directives that are not known by the current version of UniC10 Plugin. Solve by updating the UniC10 Plugin DLL to the latest version.
7	Transponder type specified is unknown because it was introduced by a later version of UniC10. Solve by updating the UniC10 Plugin DLL to the latest version.
8	Commission file invalid.
9	The commission cannot be used because it defines external data source fields. UniC10 Plugin currently does not support external data sources.

6.5 Legic®

Optional functions for handling Legic® master data. Note that the functions in this section are completely optional and only have to be called if the default IAM dialog has been disabled in the *PrepareCoding* function (by setting the *parent* parameter to 0).

6.5.1 ReadLegicMaster

int ReadLegicMaster()

This function reads a Legic® master card (IAM, GAM, SAM, XAM_1) and adds it to the reader's master data memory. This function can only be called in the "Online" or "Offline" state while there is a reader connected (see function *GetCodingState*).

In	
Out	
Return	
0	Master card successfully added to master data memory.
1	The connected reader does not support Legic® media.
2	No card or no Legic® card in the RFID field of the reader.
3	Invalid coding state. Has to be "Online" or "Offline" and a reader has to be connected (see function <i>GetCodingState</i>).
4	Could not add Legic® master data. The reason can be for example that the card is no master card, the master data was already added or the master data memory of the reader is full.

6.5.2 CheckLegicMasters

int CheckLegicMasters(PWCHAR &missingMasters)

This function checks which Legic® master cards (IAM, GAM, SAM, XAM_1) are missing from the reader's memory in order to be able to encode the loaded commission(s). This function can only be called in the "online" state (see function *GetCodingState*).

In order to avoid memory leaks, the *missingMasters* pointer should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In

Out	
missingMasters	<p>A string describing the master data that will have to be added to the reader's memory before the loaded commission can be coded. Each missing master data is represented by a string of the following format: <code><slaveStamp>[ETX]<requiredMaster></code> where [ETX] is the control character 0x03. <i>slaveStamp</i> is the stamp of the segment that has no matching master loaded. <i>requiredMaster</i> is a human readable string describing what kind of master has to be added in order to encode this segment, for example "IAM <= 30000000". This human readable string is the same string that is displayed in the missing-master-dialog that appears when the <i>PrepareCoding</i> function is called with the <i>parent</i> parameter != 0.</p> <p>Multiple string descriptions (for multiple required masters) are separated by an STX (0x02) character. The last string description is also terminated by an STX character.</p> <p>If more the than one Legic® commission is loaded for combi coding, this list contains all missing masters of all Legic® commissions.</p>
Return	
0	Execution successful.
1	The connected reader does not support Legic® media.
2	None of the loaded commissions is a Legic® commission.
3	Invalid coding state. Has to be "Online" (see function <i>GetCodingState</i>).
4	Error accessing the reader's master data memory.

6.5.3 ReadLegicMasterMemory

int ReadLegicMasterMemory (PWCHAR &masters)

This function returns the contents of the reader's Legic® master card memory. It can be used to check if certain masters are already loaded in the reader's memory.

This function only works if the connected reader is one of the following readers:

- Legic Prime MSM-S
- Legic Advant SM2570C
- Legic Advant SM4500

In order to avoid memory leaks, the *masters* pointer should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In	
Out	
masters	<p>A string describing the reader's master data memory. Each master data is represented by a hex string of its stamp.</p> <p>Multiple masters are separated by an STX (0x02) character. The last string master data is also terminated by an STX character.</p> <p>For example, if a reader contains two masters, the returned string might look like this: "3000ABCD[STX]28289C450032E4[STX]".</p>
Return	
0	Execution successful.
1	There is no Legic® reader connected.
2	Error accessing the reader's master data memory.

6.5.4 ReadLegicAdvantStamp

int ReadLegicAdvantStamp (BYTE segNum, PWCHAR &stamp)

This function returns the stamp of the specified segment of a Legic® Advant card which is present on the reader.

In order to avoid memory leaks, the *stamp* pointer should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In	
segNum	The number of the segment whose stamp should be read.
Out	
stamp	The stamp of the segment with the specified number in hex format.
Return	
0	A stamp could be read successfully.
1	No reader or no compatible reader connected.
2	Invalid coding state. Has to be "Offline" or "Online" (see function <i>GetCodingState</i>).
3	No Legic Advant card or more than one Legic Advant card found on the reader.

4	There is no segment with the specified number or the specified segment is read protected.
---	---

6.6 Coding - asynchronous

Functions for performing the card by card coding and handling the coding state. This section describes the asynchronous coding approach which automatically performs the coding in the background. On the downside, a relatively large number of methods has to be implemented and a deeper understanding of the internal coding state machine is necessary.

If the threading should be handled on the client side, it is recommended to use the synchronous coding approach described in the following chapter.

6.6.1 Code

int Code(PWCHAR cardID, PWCHAR varData)

This function starts the asynchronous coding process with the supplied user-definable data. The function returns immediately and the actual coding state is reported through window messages to the window specified in the *codingCallback* parameter of the associated *PrepareCoding* function.

Please note that this function can only be used if callback mode is not activated (i.e. the *codingCallback* parameter of *PrepareCoding* was set to a value other than 0). If callback mode is activated, use the function *CodeWithCallback* instead.

In	
cardID	The decimal card ID that is to be coded into the commission's card ID field in string representation. If, for example, the card ID 123 (decimal) is to be coded, this parameter has to be the string "123". If the commission specifies no card ID, this field is ignored.
varData	<p>A list separated by STX characters (0x02) of the variable data of the commission. Each variable data contains the variable area name, an ETX character [0x03] and the actual data which is to be written. The format of the actual data depends on the type of variable data field.</p> <p>Example: A commission contains four variable data fields:</p> <ol style="list-style-type: none">1. "Year of birth", Input Format: Decimal2. "Currency", Input Format: Hex3. "Name", Input Format: ASCII4. "CCN", Input Format: BCD <p>A valid varData string for this commission would be:</p> <p>"Year of birth[ETX]1975[STX]Currency[ETX]D320[STX]Name[ETX]Paul Smith [STX]CCN[ETX]76544457[STX]" where [STX] is the character code 0x02 and [ETX] 0x03. The last varData entry is also terminated by [STX].</p>

Out	
Return	
0	Coding started.
1	Invalid card ID. Either no decimal value or doesn't fit into the commission's smallest card ID field.
2	Invalid var data string.
3	Invalid coding state. Has to be "Online" (see function <i>GetCodingState</i>).
4	License invalid. This can occur for example if a dongle license is in use and the dongle has been removed. The internal coding state however is not changed so if the dongle is reinserted again, the <i>Code</i> function will succeed again.
5	Invalid coding mode. Callback mode was selected by setting the <i>codingCallback</i> parameter of function <i>PrepareCoding</i> to 0.

6.6.2 SleepAndCode

int SleepAndCode(PWCHAR cardID, PWCHAR varData, int searchSleep)

This function is similar to the *Code* function. The additional parameter searchSleep sets a sleep time by which the coding is delayed after a card is detected and before the coding state is switched to "CreateCard". This is necessary for some printers which do not immediately reach the coding position of a card but instead move the card back out of the field before the final coding position is reached. Without the delay the coding would start as soon as the card enters the field – because the card is still in movement however it might leave the field again while the coding is still in progress which would cause a coding failure.

In	
cardID	See <i>Code</i> function
varData	See <i>Code</i> function
searchSleep	Time in milliseconds by which the coding is delayed after a card is found and before the coding state is switched to "CreateCard"
Out	
Return	
See <i>Code</i> function	

6.6.3 CodeWithCallback

int CodeWithCallback(PWCHAR cardID, PWCHAR varData, int searchSleep, TCALLBACK callback)

This function is to be used instead of the *Code* (or *SleepAndCode*) function when callback was activated (by passing 0 as the *codingCallback* parameter of the *PrepareCoding* function).

The main difference to the *Code* function is, that the status callbacks of the coding are not sent via the windows message WM_CREATE_CARD (0x804D). Instead, the status callbacks are passed to the function that is referenced by the callback parameter of this function. The callback function has to a function of the following type:

void Callback(WPARAM wParam, LPARAM lParam)

wParam and *lParam* correspond directly to the according windows message parameters as described in the *PrepareCoding* function description (parameter *codingCallback*).

In	
cardID	See <i>Code</i> function
varData	See <i>Code</i> function
searchSleep	See <i>SleepAndCode</i> function
callback	The callback function that receives the asynchronous status callbacks of the coding process.
Out	
Return	
See <i>Code</i> function. Error code 5 is thrown by this function when callback mode is off.	

6.6.4 AbortCoding

int AbortCoding()

Attempts to abort a running coding process. This only works if the coding process is in the "scan for card" or "wait for remove" state (see function *GetCodingState*).

Note that this function may take several hundred ms to return because it may need to wait for a reader card polling cycle to finish. Once it does return though, the coding state has been changed to "online".

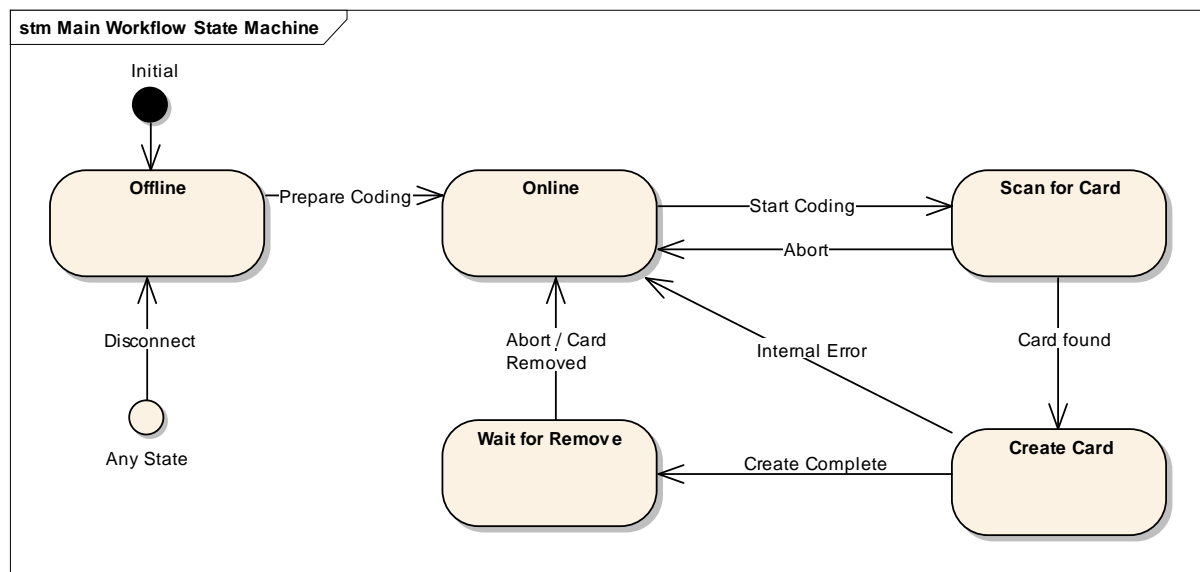
In
Out

Return	
0	Coding aborted successfully and coding state changed to "online".
1	Coding could not be aborted because it is currently not in the "scan for card" or "wait for remove" state.

6.6.5 GetCodingState

int GetCodingState()

Returns the current coding state. The following states and their transitions are implemented:



Note that the coding state is only of interest if the coding is performed asynchronously. The synchronous coding only knows the states Offline and Online and discards the other states.

In	
Out	
Return	
0	Offline – coding has not yet been initialized with a valid commission and reader.
1	Online – coding is initialized and can be started.
2	Scan for card – coding has been started by the Code function and the reader is currently searching for a card that can be coded.
3	Create card – card is currently being coded.
4	Wait for remove – card was coded but is still in the reader's field.

6.6.6 GetLog

int GetLog(LPPARAM logHandle, PWCHAR &cardUID, int &createdDate, BYTE &codingResult, PWCHAR &shortDesc, PWCHAR &cardID, PWCHAR &varData)

Resolves the log handle which can be obtained by the callback message of the *Code* function into actual logging data.

The client application can decide whether to store the log, just use it to display a coding's outcome or ignore it completely. In any case, to avoid memory leaks, the logHandle should be freed with the *FreeLog* function at some time after it has been obtained.

Note that if a combi coding was performed, only the information of the last chip is returned by this function. This is sufficient if only the coding result has to be obtained because the success of the last coded chip is also the success of the whole coding. If logs of each individual card have to be obtained, the function *GetCombiLog* has to be called.

Furthermore, the *cardUID*, *shortDesc*, *cardID* and *varData* pointers should be freed by passing them to the *FreeWideChar* function once they are not needed anymore.

In	
logHandle	Handle to the log data as obtained by the WM_CREATE_CARD callback message (see <i>PrepareCoding</i> function). The coding states CODING_SUCCESS and CODING_FAIL both contain log data in their lParam.
Out	
cardUID	A hex string containing the UID (serial number) of the coded card. The length depends on the transponder type of the card.
createdDate	Date and time when the coding of the card finished. The time is represented as seconds since the standard epoch of 1/1/1970 00:00:00 (Unix timestamp).
codingResult	The outcome of the coding. TRUE (>0) values are to be interpreted as a successful coding, FALSE (==0) values as a failed coding.
shortDesc	A short description that serves as a coding protocol overview. It usually contains the user-definable data (card ID and varData) that was used for the card. If the coding failed, this field contains the error message that caused the coding to fail.
cardID	The decimal card ID that was coded into the commission's card ID field in string representation.

varData	<p>A list separated by STX characters (0x02) of the variable data that was coded into the card. Each variable data contains the variable area name, an ETX character [0x03] and the actual data which was written. The format of the actual data depends on the type of variable data field.</p> <p>Example: A commission contains four variable data fields:</p> <ol style="list-style-type: none"> 1. "Year of birth", Input Format: Decimal 2. "Currency", Input Format: Hex 3. "Name", Input Format: ASCII 4. "CCN", Input Format: BCD <p>A possible varData string for a coded card could be:</p> <p>"Year of birth[ETX]1975[STX]Currency[ETX]D320[STX]Name[ETX]Paul Smith[STX]CCN[ETX]76544457" where [STX] is the character code 0x02 and [ETX] 0x03.</p>
Return	
0	OK
1	Invalid log handle.

6.6.7 GetCombiLog

int GetCombiLog(LPPARAM logHandle, BYTE combiIndex, PWCHAR &cardUID, int &createdDate, BYTE &codingResult, PWCHAR &shortDesc, PWCHAR &cardID, PWCHAR &varData)

An extension of the *GetLog* function that should be used if information on combi coded cards is to be obtained. The only difference to the *GetLog* function is the added *combiIndex* parameter.

In	
logHandle	See <i>GetLog</i> function.
combiIndex	Specifies for which of the combi coded chips the log information is returned. The coding that corresponds to the first commission loaded in <i>PrepareCoding</i> has index 0, the second index 1, etc.
Out	
cardUID	See <i>GetLog</i> function.
createdDate	See <i>GetLog</i> function.
codingResult	See <i>GetLog</i> function.
shortDesc	See <i>GetLog</i> function.
cardID	See <i>GetLog</i> function.

varData	See <i>GetLog</i> function.
Return	
0	OK
1	Invalid log handle.
2	Invalid combi index.

6.6.8 FreeLog

int FreeLog(LPARAM logHandle)

Frees the log handle and the memory allocated by the actual log data. This function should be called for all log handles that are retrieved through the callback messages of the *Code* function.

In	
logHandle	Handle to the log data as obtained by the WM_CREATE_CARD callback message (see <i>PrepareCoding</i> function. The coding states CODING_SUCCESS and CODING_FAIL both contain log data in their lParam.
Out	
Return	
0	Log handle and associated log data freed successfully.
1	Invalid log handle.

6.7 Coding – synchronous

Version 4.16.0 of UniC10 Plugin DLL introduced a new and simpler approach to perform the card-by-card coding: Synchronous coding. For this approach, only the implementation of one blocking coding function is necessary. The threading should be done on the client side when using this approach to avoid that long running coding operations block the user interface.

Additionally, combi codings are not supported for synchronous coding operations. So, if more than one chip has to be encoded at once, the client has to handle the combi coding itself (by calling *PrepareCoding* + *SyncCode* for every chip in a card).

6.7.1 SyncCode

int SyncCode(PWCHAR cardID, PWCHAR varData, PWCHAR &cardUID, PWCHAR &cardIDAfterCode, PWCHAR &errorMessage)

Performs the coding with the supplied user-definable data (card ID and vardata). This function returns after the coding is finished.

In	
cardID	The decimal card ID that is to be coded into the commission's card ID field in string representation. If, for example, the card ID 123 (decimal) is to be coded, this parameter has to be the string "123". If the commission specifies no card ID, this field is ignored.
varData	<p>A list separated by STX characters (0x02) of the variable data of the commission. Each variable data contains the variable area name, an ETX character [0x03] and the actual data which is to be written. The format of the actual data depends on the type of variable data field.</p> <p>Example: A commission contains four variable data fields:</p> <ol style="list-style-type: none">5. "Year of birth", Input Format: Decimal6. "Currency", Input Format: Hex7. "Name", Input Format: ASCII8. "CCN", Input Format: BCD <p>A valid varData string for this commission would be:</p> <p>"Year of birth[ETX]1975[STX]Currency[ETX]D320[STX]Name[ETX]Paul Smith [STX]CCN[ETX]76544457[STX]" where [STX] is the character code 0x02 and [ETX] 0x03. The last varData entry is also terminated by [STX].</p>

Out	
cardUID	The UID of the encoded card.
cardIDAAfterCode	The card ID of the encoded card after the coding – note that in the most cases, this value will be the same as the input card ID. However, there are special cases where the encoding file reads the card ID from an existing data structure and is therefore only know after the coding.
errorMessage	An error message describing the coding error in case the coding failed.
Return	
0	OK
1	Invalid card ID. Either no decimal value or doesn't fit into the commission's smallest card ID field.
2	Invalid var data string.
3	Invalid coding state. Has to be "Online" (see function <i>GetCodingState</i>).
4	License invalid. This can occur for example if a dongle license is in use and the dongle has been removed. The internal coding state however is not changed so if the dongle is reinserted again, the <i>Code</i> function will succeed again.
5	Combi coding file loaded in <i>PrepareCoding</i> function. Combi codings are not supported by the synchronous code function.
6	Either none or more than one card of the transponder type specified in the coding file found in the RFID field of the reader.
7	Coding error. The error message that led to the failure of the coding is stored in the <i>errorMessage</i> result parameter.

6.7.2 SyncCodeExtraInfo

int SyncCodeExtraInfo(PWCHAR cardID, PWCHAR varData, PWCHAR &cardUID, PWCHAR &transponderTypeAfterCode, PWCHAR &userInput, int &chipType, int &chipVersion, PWCHAR &laserParametersFront, PWCHAR &laserParametersBack, PWCHAR &errorMessage)

Performs the coding with the supplied user-definable data (card ID and vardata). This function returns after the coding is finished. In addition to the function *SyncCode*, this function returns additional information about the coded commission.

In	
cardID	The decimal card ID that is to be coded into the commission's card ID field in string representation. If, for example, the card ID 123 (decimal) is to be coded, this parameter has to be the string "123". If the commission specifies no card ID, this field is ignored.
varData	<p>A list separated by STX characters (0x02) of the variable data of the commission. Each variable data contains the variable area name, an ETX character (0x03) and the actual data which is to be written. The format of the actual data depends on the type of variable data field.</p> <p>Example: A commission contains four variable data fields:</p> <ul style="list-style-type: none"> • "Year of birth", Input Format: Decimal • "Currency", Input Format: Hex • "Name", Input Format: ASCII • "CCN", Input Format: BCD <p>A valid varData string for this commission would be:</p> <p>"Year of birth[ETX]1975[STX]Currency[ETX]D320[STX]Name[ETX]Paul Smith [STX]CCN[ETX]76544457[STX]" where [STX] is the character code 0x02 and [ETX] 0x03. The last varData entry is also terminated by [STX].</p>

Out	
cardUID	The UID of the encoded card.
Transponder-TypeAfter-Code	The transponder type after coding. In the most cases, this is the same value as returned for example in the GetLoadedCommissionTransType function, for example the Q5/T55x7 type however can be changed to a different type like EM4102 or HIDProx during coding.
userInput	<p>A string containing all variable fields which were set during the coding, the so-called user inputs. Multiple user inputs are separated by an STX character (0x02) and each user input is a 4-tuple of the following fields, separated by ETX (0x03):</p> <ul style="list-style-type: none"> • type: One of CARDIDNUMBER, VARDATA or EXTDATASRC • name: The name of the variable or external data source field, empty for card ID • value: The value of the user input (e.g. the card ID) • source: Either HUMAN (= manual input as function parameter) or READFROMSELF (= value determined automatically during coding) <p>An example for a user input string would be: "CARDIDNUMBER[ETX][ETX]1234[ETX]HUMAN[STX]VARDATA[ETX]PersNr[ETX]332-Q3[ETX]READFROMSELF[STX]"</p> <p>Note that in the most cases, these user inputs match what was set in the input parameters cardID and varData of this function. However, there are special cases where some variable data is set during the coding, for example when using read-from-self objects; when coding multiple commissions in a combi card, this has to be considered (for example the second coding might need the first coding output card ID as input card ID)</p>
chipType	The index of the coded transponder's chipType
chipVersion	The index of the coded transponder's chipVersion
laserParametersFront	A set of preformatted laser layout parameters which can be sent to a Rohlin Laser running VMC2 by the VAR-Command. The parameter string is a list of tuples which are separated by an STX (0x02) character. Each tuple is separated by ETX (0x03) and contains the var field name and value, for example "CARDID[ETX]1234[STX]UID[ETX]08E34521[STX]
laserParametersBack	Same as laserParametersFront, only for back side parameters
errorMessage	An error message describing the coding error in case the coding failed.

Return	
0	OK
1	Invalid card ID. Either no decimal value or doesn't fit into the commission's smallest card ID field.
2	Invalid var data string.
3	Invalid coding state. Has to be "Online" (see function <i>GetCodingState</i>).
4	License invalid. This can occur for example if a dongle license is in use and the dongle has been removed. The internal coding state however is not changed so if the dongle is reinserted again, the <i>Code</i> function will succeed again.
5	Combi coding file loaded in <i>PrepareCoding</i> function. Combi codings are not supported by the synchronous code function.
6	Either none or more than one card of the transponder type specified in the coding file found in the RFID field of the reader.
7	Coding error. The error message that led to the failure of the coding is stored in the <i>errorMessage</i> result parameter.
8	Unable to build the <i>laserParameters</i> string because of invalid settings in the coding file or unknown UID Converter ID (maybe because the UIDConverter.DLL is missing or doesn't contain the required converter).

6.8 Reading

Functions related to the optional card reading functionality.

6.8.1 LoadReadDefinition

int LoadReadDefinition(PWCHAR readDefFile)

Loads a read definition file (*.RDX) which describes the data that should be read with the *ReadCard* function.

Read definition files are created by MADA Marx Datentechnik, similar to commission files.

In	
readDefFile	Full path name of the read definition file to load.
Out	
Return	
0	OK
1	File not found.
2	File is no valid read definition file.

6.8.2 ReadCard

int ReadCard(PWCHAR &readData)

Reads the data from a transponder as defined in the read definition file loaded by the *LoadReadDefinition* function.

Since the main use of the read function is to read card numbers from transponders, the read data is always a decimal number. Note that while the returned string represents a decimal number, this number is not limited in size, so it is not guaranteed that it can be converted into a 32 bit Integer.

This function can only be called in the coding states Offline or Online. Furthermore, connection to a reader has to be established and the read definition needs to have been loaded by calling the *LoadReadDefinition* function. Also, the connected reader must support the transponder defined in the loaded read definition file.

In order to avoid memory leaks, the *readData* pointer should be freed by passing it to the *FreeWideChar* function once it is not needed anymore.

In

Out	
readData	The read data in decimal representation
Return	
0	OK
1	Wrong coding state. Must be Online or Offline.
2	No read definition file loaded.
3	No reader connected.
4	The connected reader cannot read the transponder specified in the loaded read definition file.
5	Zero or more than one transponder in the field that matches the transponder specified in the loaded read definition file.
6	The data defined in the read definition file could not be read from the transponder. Either the transponder was removed from the field before the data could be read or the coded structure on the transponder does not match the data of the read definition file.

7 Version History

1.17.1	<ul style="list-style-type: none"> Added Deployment chapter Fixed some formatting and typos
1.17.0	<ul style="list-style-type: none"> Parameter commFileName of function PrepareCoding also accepts XML file contents in addition to file names Added function SyncCodeExtraInfo DLL version reference removed from function GetDLLVersion
1.16.0	<ul style="list-style-type: none"> Added description of synchronous coding in implementation guidelines and marked registration as optional Corrected licensing information regarding self-created coding files in the Registration chapter Reorganised function chapter into subsections and added remarks about optional functionalities Applied new format template DLL version 4.16.x match to document version in function GetDLLVersion
1.15.0	<ul style="list-style-type: none"> Added function ReadLegicAdvantStamp DLL version 4.15.x match to document version in function GetDLLVersion
1.14.0	<ul style="list-style-type: none"> Added function ReadLegicMasterMemory DLL version 4.14.x match to document version in function GetDLLVersion
1.13.0	<ul style="list-style-type: none"> Added function AnalyzeCodingMask DLL version 4.13.x match to document version in function GetDLLVersion
1.12.0	<ul style="list-style-type: none"> Added description of TCP/IP connection to function ConnectReader DLL version 4.12.x match to document version in function GetDLLVersion
1.11.0	<ul style="list-style-type: none"> Added description for autoconnect mode for specific reader in function ConnectReader Added function GetConnectedComPort DLL version 4.11.x match to document version in function GetDLLVersion
1.10.0	<ul style="list-style-type: none"> Added function SearchCardsExtraInfo DLL version 4.10.x match to document version in function GetDLLVersion
1.9.1	<ul style="list-style-type: none"> Added "Offline" to allowed coding states in function ReaderLegicMaster
1.9.0	<ul style="list-style-type: none"> Added function GetReaderSNR DLL version 4.9.x match to document version in function GetDLLVersion
1.8.0	<ul style="list-style-type: none"> Added description about callback mode to PrepareCoding function's codingCallback parameter Added function CodeWithCallback Added notes about callback mode to Code function Added error code 5 to Code function DLL version 4.8.x match to document version in function GetDLLVersion
1.7.1	<ul style="list-style-type: none"> Added license file description to registration section Added license file remark to function CheckRegistration Added license file remark to function GetCustomerCode
1.7.0	<ul style="list-style-type: none"> Added autoconnect mode description to functions ConnectReader, CheckReaderConnection, DisconnectReader, PrepareCoding Added notes about autoconnect mode to implementation guidelines Added error code 2 to function CheckReaderConnection Added error code 11 to function PrepareCoding DLL version 4.7.x match to document version in function GetDLLVersion
1.6.1	<ul style="list-style-type: none"> Added note to abort coding after a timeout in implementation guidelines

1.6.0	<ul style="list-style-type: none"> • Added function LoadReadDefinition • Added function ReadCard • Added notes about the read function in overview. • Added combi coding note to GetLog function • Added combi coding note to CheckLegicMasters function • Added function GetCombiLog • Added comments about combi coding to GetLoadedCommissionTransType function • Added combi functions to PrepareCoding function • Added remark about combi coding to overview • Added notes about combi coding in implementation guidelines • DLL version 4.6.x match to document version in function GetDLLVersion
1.5.1	<ul style="list-style-type: none"> • Added remark about resetting the timeout of Legic® master data memory in function CheckReaderConnection.
1.5	<ul style="list-style-type: none"> • Added function CheckLegicMasters • Added function SearchCards • Added function GetLoadedCommissionTransType • Added function FreeWideChar • Added examples of using CheckLegicMasters and ReadLegicMasters to Implementation Guidelines • Added chapters “Registration” and “Memory Management” • Added notes about FreeWideChar to functions that return PWCHAR pointers • Added error code 9 to function PrepareCoding • DLL version 4.5.x match to document version in function GetDLLVersion
1.4	<ul style="list-style-type: none"> • Added 0 value description to parent parameter of function PrepareCoding • Added function ReadLegicMaster • DLL version 4.4.x match to document version in function GetDLLVersion
1.3	<ul style="list-style-type: none"> • Added function SleepAndCode • DLL version 4.3.x match to document version in function GetDLLVersion
1.2	<ul style="list-style-type: none"> • Added function IsPCSCReader • Added PCSC description to ConnectReader function • DLL version 4.2.x match to document version in function GetDLLVersion
1.1	<ul style="list-style-type: none"> • Fixed parameter description of PrepareCoding function (in / out section headers incorrect) • Added return value 4 (License invalid) in Code function • Added return values 4 to 8 in PrepareCoding function (License invalid and more precise error codes for commission file parsing) • Changed return value 1 of PrepareCoding function “Invalid commission file or file not found.” -> “File not found” • Added function SetLanguage • Added function CheckRegistration • Added function PerformRegistration • Added function ClearRegistration • Added function GetCustomerCode • DLL version 4.1.x match to document version in function GetDLLVersion • Added description of registration process and customer codes • Added registration step to implementation guidelines
1.0	Initial Revision