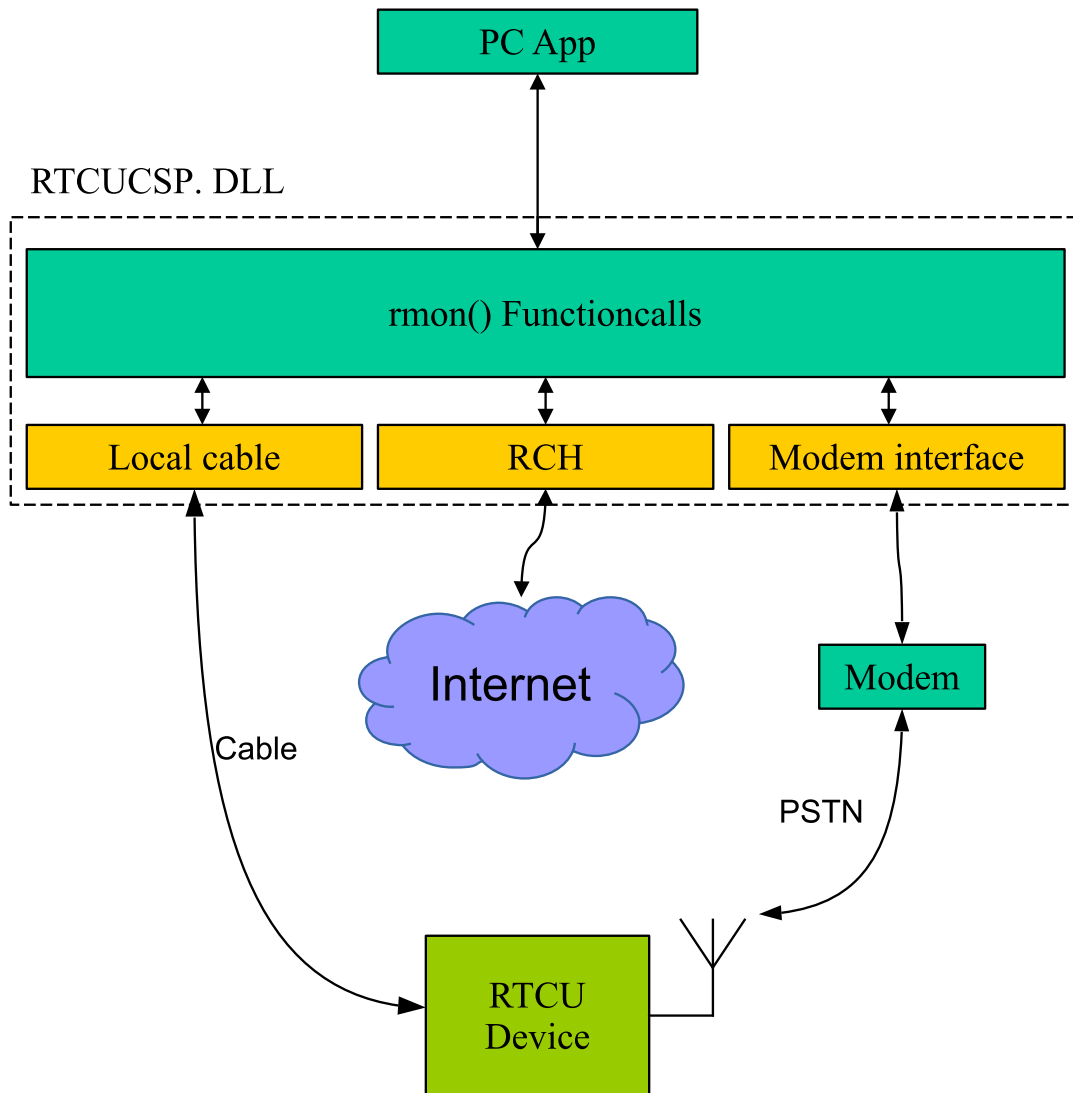


# RTCUCSP. DLL

## RTCUCSP. DLL

### Version 4.60



## Table of Contents

Introduction .....	7
Graphic illustration of the library .....	8
Contents of package .....	9
Interface and state diagram .....	10
Initializing the library .....	11
Entering the Idle State .....	11
Initializing the connection .....	11
Entering the connection Idle State .....	11
The Local/Remote Connected State .....	11
Closing or changing the connection .....	12
Functions in the RTCUCSP.DLL library .....	13
Return codes .....	13
Initialization/Configuration .....	14
rmonOpen () .....	14
rmonClose() .....	14
rmonGetVer() .....	14
rmonSetMaxConnections() .....	14
rmonSetGWParameters() .....	15
rmonSetGWParametersAdv() .....	15
rmonSetRCHParametersSecure() .....	16
rmonGetPortList() .....	16
rmonEnumeratePorts() .....	16
rmonOpenConnection() .....	17
rmonCloseConnection() .....	17
rmonSetComport() .....	17
rmonSetModemInit() .....	18
rmonSetRemoteBaudrate() .....	18
rmonConnect() .....	18
rmonDisconnect() .....	19
rmonConnected() .....	19
rmonAuthenticate() .....	19
rmonEnableLargeData() .....	20
Program/Firmware upload .....	21
rmonFirmwareUpload() .....	21
rmonFirmwareStartUpload() .....	22
rmonFirmwareResumeUpload() .....	23

rmonApplicationUpload()	23
rmonApplicationStartUpload()	24
rmonApplicationResumeUpload()	25
rmonVoiceUpload()	25
rmonNumOfVoiceMessages()	26
rmonCheckTransfer()	26
rmonApplicationFilename()	27
Manipulation of Persistent memory	28
rmonPersistentRead()	28
rmonPersistentWrite()	29
rmonReadPersistentFRAM()	30
rmonWritePersistentFRAM()	30
rmonReadPersistentFLASH()	31
rmonWritePersistentFLASH()	31
rmonGetXFLASHSize()	32
rmonReadPersistentXFLASH()	32
rmonWritePersistentXFLASH()	33
Datalogger	34
rmonLogFirst()	34
rmonLogLast()	34
rmonLogReadExt()	35
rmonLogGetValuesPerRecord()	35
rmonLogClear()	36
rmonLogGotoLinsec()	36
rmonLogReadByTag()	37
rmonLogSeek()	37
I/O system functions	38
rmonReadIOMemory()	38
rmonWriteIOMemory()	39
rmonGetIOState()	39
rmonSetIOState()	40
rmonGetIOCount()	40
Real time clock	41
rmonGetRTC()	41
rmonSetRTC()	41
GSM/SMS functions	42
rmonGetIMEI()	42
rmonGetIMSI()	42
rmonGetICCID()	42

rmonSendSMS()	43
rmonReceiveSMS()	43
rmonReceiveSMSEnable()	44
rmonGetGSMSignalLevel()	44
rmonSetAllowedCallerList()	44
rmonGetAllowedCallerList()	45
rmonSetGSMPIN()	45
rmonGetGSMPIN()	45
Filesystem functions	46
rmonMediaPresent()	46
rmonMediaWriteprotected()	47
rmonMediaOpen()	47
rmonMediaClose()	47
rmonMediaQuickformat()	48
rmonMediaQuickformatX()	48
rmonMediaEject()	48
rmonMediaInformation()	49
rmonMediaSize()	49
rmonFSStatusLED()	50
rmonDirCreate()	50
rmonDirChange()	50
rmonDirCurrent()	50
rmonDirCatalog()	51
rmonDirCatalogX()	51
rmonDirDelete()	52
rmonFileCreate()	52
rmonFileOpen()	52
rmonFileExists()	53
rmonFileRename()	53
rmonFileDelete()	53
rmonFileStatus()	54
rmonFileGetInfo()	54
rmonFileSeek()	54
rmonFilePosition()	55
rmonFileRead()	55
rmonFileReadString()	55
rmonFileWrite()	56
rmonFileWriteString()	56
rmonFileWriteStringNL()	56

rmonFileClose().....	57
rmonFileFlush().....	57
Security functions .....	58
rmonSecurityImport() .....	58
rmonSecurityRemove().....	58
rmonSecurityInfo().....	58
System Object Storage functions .....	59
rmonObjectRead().....	59
rmonObjectReadX().....	60
rmonObjectWrite().....	61
rmonObjectWriteX() .....	62
rmonObjectErase().....	63
Misc. functions.....	64
rmonReset() .....	64
rmonResetX().....	64
rmonHalt() .....	64
rmonGetSerialNumber().....	65
rmonVer() .....	66
rmonSetPassword() .....	66
rmonGetTargetInfo().....	67
rmonGetTargetProfile() .....	68
rmonGetDeviceInfo().....	69
rmonReceiveDebugMsg().....	70
rmonGetDebugEnabled().....	70
rmonSetDebugEnabled() .....	70
rmonVoiceMessagesAbove64K().....	70
rmonGetAppInfo() .....	71
rmonGetGPRSSettings() .....	71
rmonSetGPRSSettings().....	72
rmonGetGatewaySettings() .....	73
rmonSetGatewaySettings().....	74
rmonRCHGetConfig() .....	75
rmonRCHSetConfig().....	76
rmonRCHGetAutoConnect().....	76
rmonRCHSetAutoConnect() .....	77
rmonGetLANSettings().....	77
rmonSetLANSettings() .....	78
rmonGetWLANSettings() .....	79
rmonSetWLANSettings().....	80

rmonFaultLogRead() .....	81
rmonFaultLogReadX() .....	82
rmonFaultLogClear() .....	82
rmonFaultGetText() .....	83
rmonSoftwareUpgrade() .....	83
rmonFlexOption() .....	83
rmonStatisticsRead() .....	84
rmonGetUnitState() .....	85
rmonGetPowerInformation() .....	85
Appendix A, simple application .....	87
Appendix B, RTCUPROG application .....	89

## Introduction

This document describes the RTCU CSP (Communication Support Package), an API library of functions that allows communication with RTCU products on the same level of functionality available when using the RTCU M2M Studio.

Establishing a connection to the RTCU device can be done over a direct cable connection or remotely over the RTCU Communication Hub (RCH). Using CSD (modem) is also supported for devices that support this older connection technology.

For more information on the RACP protocols used by the CSP, please refer to the: "RTCU Communication Protocol Documentation Set."

This document also contains the source code for a simple application (Appendix A) and a complete application with the RTCU Programming Tool application (Appendix B).

The RTCU Programming Tool is a full-featured application that can be downloaded as a fully installable package that allows uploading applications and firmware as well as setting up various device parameters.

The RTCU Programming Tool shows all the different aspects of establishing and maintaining an application with an RTCU device, authentication, etc.

Both applications will give an excellent hands-on experience to the library and function as a good starting point for application development.



## Contents of package

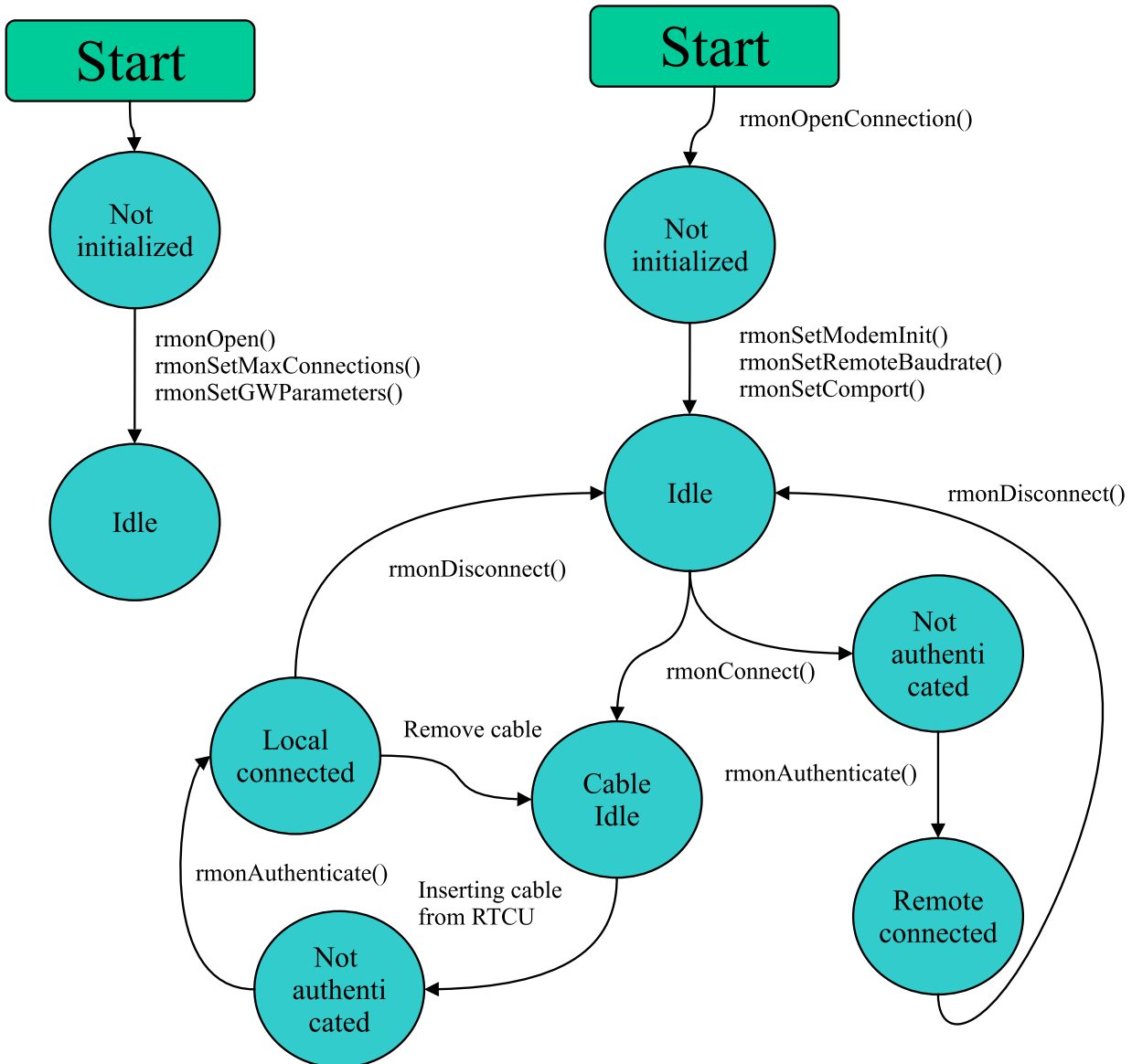
The package this document is part of, contains the following:

"\RTCU Communication Support Package.pdf"	This document
"\RTCUPROG"	The RTCUPROG application described in appendix B.
"\Demo"	A simple 32/64 bit Demo application.
"\Library"	The library root folder.
"\Library\H"	Library header file folder.
"\Library\LIB32"	Library for 32-bit applications.
"\Library\LIB64"	Library for 64-bit applications.
"\Library\DLL32"	DLL files for 32-bit applications.
"\Library\DLL64"	DLL files for 64-bit applications.

Microsoft Visual Studio C++ 2019 will be needed to use and build this library.

## Interface and state diagram

To be able to go into details about using the interface, it is necessary to know the different states in which the communication protocol can be:



## Initializing the library

To begin with, you are in the **Not Initialized State**. To proceed from there, you will have to carry out the following two steps:

- **Step A.**  
You will have to **prepare the communication system** with `rmonSetGWParameters` and `rmonSetMaxConnections`
- **Step B.**  
You will have to **open the communication system** with `rmonOpen()`.

## Entering the Idle State.

After the communication system is initialized, the **Idle State** is entered.  
From there, it is possible to open connections to RTCU devices.

## Initializing the connection.

After calling the `rmonOpenConnection`, you are in the **Not Initialized State**. To proceed from there, you will have to carry out the following two steps:

- **Step A.**  
Then determine **what kind of connection** you want to be started.
  - Should it be a local cable connection?
  - Should it be a data call (CSD) modem connection?
  - Should it be an RTCU Communication Hub connection?
- **Step B.**  
You will have to **prepare the connection** with `rmonSetComport()`, `rmonSetModemInit()` and `rmonSetRemoteBaudrate()` if connection is local cable or CSD modem.

## Entering the connection Idle State.

After the connection is initialized, the **Connection Idle State** is entered.  
From there it is possible to connect to an RTCU device either through

- A local cable connection or
- A remote connection – (modem (CSD) or RTCU Communication Hub connection)

For remote connections calling `rmonConnect()` will enter the **Not Authenticated State**.

For cable connection, calling `rmonConnected` will enter the **Cable Idle State**, and from here, inserting a cable between the PC and RTCU device will enter the **Not Authenticated State**.

From the **Not Authenticated State** carry out the `rmonAuthenticate()`, which will put the library into either the **Locally Connected State** or the **Remotely Connected State**.

## The Local/Remote Connected State.

The communication is now up running, and your PC application can now communicate with the RTCU device using the functions described.

## Closing or changing the connection.

### A. The Locally Connected State:

As you can see in the state diagram above, you have **three possibilities of** being in a **Locally Connected State**. When you want to leave this state, you can either

- Remove the cable, which will bring you into the **Cable Idle State**.
- Use `rmonDisconnect()`, which will bring you into the **Connection Idle State**.
- Close the connection (using `rmonCloseConnection`).

### B. The Remotely Connected State:

As you can see in the state diagram above, you have **three possibilities of** being in a **Remotely Connected State**. When you want to leave this state, you can either

- Use `rmonDisconnect()`, which will bring you into the **Connection Idle State**.
- Close the connection (using `rmonCloseConnection`).

## Functions in the RTCUCSP.DLL library

In the following description of the different function calls, we will differentiate between NX32L, NX32 and X32 RTCU devices, as some of the functions are not supported on some models.

Please consult Logic IO's website for up-to-date information on new products and their availability.

### Return codes

The return codes from most of the functions will be one of those shown below. These return codes are declared in the header file (rtcucsp.h) for the library as an "enum struct rmonRet".

Symbolic name	Value	Description
rmonOK	0	Operation OK
rmonError	1	General error
rmonComError	2	Communication error
rmonTargetError	3	Other error in device than rmonComError
rmonIllegalHandle	4	The connection handle is illegal (corrupt/non-existent)
rmonIllegalTarget	5	Target and type of firmware file does not match
rmonOnlyGateway	6	The function can only be used via the RCH
rmonNotGateway	7	The function can not be used over the RCH
rmonDenied	8	Access to device denied
rmonNoData	9	No data
rmonNoMoreData	10	No more data
rmonNotInit	11	Not initialized
rmonInit	12	The RTCUCSP library is already initialized
rmonConnection	13	There is a connection established already
rmonGatewayNotFound	14	RCH not found
rmonFileNotFound	15	Application or Firmware file not found.
rmonIllegalFile	16	File specified is illegal (corrupt)
rmonOldFormat	17	Old firmware file format, not supported
rmonNoMonitorMode	18	Not able to enter monitor mode
rmonErrorReset	19	Not able to reset RTCU
rmonErrorHalt	20	Not able to halt RTCU
rmonNoBackground	21	Background transfer not supported by the RTCU device.
rmonInterrupted	22	Background transfer was interrupted.
rmonCancelled	23	Data transfer has been canceled.
rmonNotProgrammable	24	Attempt to program a Micro device with a VSX file
rmonNoModem	25	No remote serial port selected or port in use.
rmonNoCable	26	No local serial port selected or port in use.
rmonMemoryConfig	27	Memory configuration is different in Project and RTCU device.
rmonImageTooLarge	28	There are no room for the image in the RTCU device.
rmonNotModem	29	The function can not be used over Modem connection.
rmonIllegalAccess	30	The function is not accessible.
rmonDowngrade	31	The device does not support this firmware version
rmonImageSupport	33	The RTCU device does not support the image.

## Initialization/Configuration

Functions that are usually needed before any real communication can be started with the RTCU device are listed here. For the proper sequence of calling the functions, please refer to the State diagram, and to the demo applications delivered as part of this package (see appendix A and appendix B).

<b>rmonOpen ()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised
<b>Synopsis</b>	rmonRet __stdcall rmonOpen(void)	
<b>Description</b>	Opens and initializes the communication system. The system can be closed again by calling rmonClose().	
<b>Returns</b>	rmonOK, rmonError, rmonInit	

<b>rmonClose()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Idle State
<b>Synopsis</b>	rmonRet __stdcall rmonClose(void)	
<b>Description</b>	Closes communication system. The communication system must be opened with rmonOpen() in order for it to be used again.	
<b>Returns</b>	rmonOK, rmonNotInit	

<b>rmonGetVer()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised and Idle State
<b>Synopsis</b>	int __stdcall rmonGetVer(void)	
<b>Description</b>	Retrieve the version number of the RTCUCSP library.	
<b>Returns</b>	Library version scaled by 100.	

<b>rmonSetMaxConnections()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised
<b>Synopsis</b>	rmonRet __stdcall rmonSetMaxConnections (int max_sessions)	
<b>Description</b>	Set maximum number of simultaneous connections possible. If this function is not called the default number of simultaneous connections will be used.	
<b>Input</b>	<b>max_sessions</b>	The maximum number of connections. (Default: 100)
<b>Returns</b>	rmonOK, rmonInit	

<b>rmonSetGWParameters()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised
<b>Synopsis</b>	rmonRet __stdcall rmonSetGWParameters(const unsigned short Port, const unsigned long MyNodeID, const char* IP, const char* Key)	
<b>Description</b>	If a connection to a remote RTCU is to be done using the RTCU Communication Hub, some parameters have to be set before this is possible. These parameters are set using this function. Please see the RTCU M2M Studio online help for a description of these parameters (Menu: Device -> Connection -> RTCU Communication Hub).	
<b>Input</b>	<b>Port</b>	RCH port (please refer to rmonConnect() for making a connection through the RCH)
	<b>MyNodeID</b>	Node ID (can be set to 0, this will allow the RCH to issue a "dynamic" ID to this node).
	<b>IP</b>	IP address of the RCH.
	<b>Key</b>	Key value (must be set to the same value as set in the RCH)
<b>Returns</b>	rmonOK, rmonInit	

<b>rmonSetGWParametersAdv()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised
<b>Synopsis</b>	rmonRet __stdcall rmonSetGWParametersAdv(unsigned char CryptKey[16], unsigned char gw_max_connection_attempt, unsigned char gw_max_send_req_attempt, unsigned short gw_response_timeout, unsigned short gw_alive_freq)	
<b>Description</b>	This function is used to set the advanced parameters for connecting to a remote RTCU device using the RCH Please see the RTCU M2M Studio online help for a description of these parameters (Menu: Device -> Connection -> RTCU Communication Hub).	
<b>Input</b>	<b>CryptKey</b>	Encryption key. To use the default key, set all 16 bytes to 0 (zero).
	<b>gw_max_connection_attempt</b>	Maximum connection attempts. Default value: 3. Range: 1-60
	<b>gw_max_send_req_attempt</b>	Maximum transmission attempts. Default value: 3. Range: 1-60
	<b>gw_response_timeout</b>	Response timeout. Default value: 30. Range: 5-60
	<b>gw_alive_freq</b>	Keep alive frequency. Default value: 60. Range: 0-60000
<b>Returns</b>	rmonOK, rmonInit	

<b>rmonSetRCHParametersSecure()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised
<b>Synopsis</b>	rmonRet __stdcall rmonSetRCHParametersSecure(char* server, char* cert, char* key, char* password)	
<b>Description</b>	This function is used to set the parameters for making a secure connecting to a remote RTCU device using the RTCU Communication Hub. Please see the RTCU M2M Studio online help for a description of these parameters (Menu: Device -> Connection -> RTCU Communication Hub).	
<b>Input</b>	<b>Server</b>	The location of the root CA certificate to validate the server certificate.
	<b>Cert</b>	The location of the client certificate of the CSP.
	<b>Key</b>	The location of the private key of the client certificate.
	<b>Password</b>	The password for the private key.
<b>Returns</b>	rmonOK, rmonInit, rmonError	

<b>rmonGetPortList()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised and Idle State
<b>Synopsis</b>	rmonRet __stdcall rmonGetPortList (char name[RMON_MAXPORTS][9], int* size)	
<b>Description</b>	Retrieve the names and number of comports available at start up.	
<b>Output</b>	<b>Name</b>	Array of ASCII strings with the names of the comports.
	<b>size</b>	The number of comports present.
<b>Returns</b>	rmonOK, rmonNoData	

<b>rmonEnumeratePorts()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> All states
<b>Synopsis</b>	rmonRet __stdcall rmonEnumeratePorts (rmoncbserialport cbFunc, void *arg, int *count)	
<b>Description</b>	Enumerate the serial ports currently present.	
<b>Input</b>	<b>cbFunc</b>	Function that is called with the name and description of a serial port.
	<b>arg</b>	A user defined argument that is included in the callback function.
<b>Output</b>	<b>count</b>	The number of serial ports enumerated.
<b>Returns</b>	rmonOK	
	The call-back function is defined as follows: <b>typedef void (__stdcall *rmoncbserialport)(void *arg, const char *name, const char *desc);</b>	

<b>rmonOpenConnection()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Idle State
<b>Synopsis</b>	HRMONCON __stdcall rmonOpenConnection (void)	
<b>Description</b>	Open a new connection session.	
<b>Returns</b>	Handle to connection or HRMONCON_ILLEGAL if no connection.	

<b>rmonCloseConnection()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised and Idle State		
<b>Synopsis</b>	rmonRet __stdcall rmonCloseConnection (HRMONCON hCon)			
<b>Description</b>	Close a connection session.			
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> </table>		<b>hCon</b>	Handle to connection
<b>hCon</b>	Handle to connection			
<b>Returns</b>	rmonOK, rmonIllegalHandle			

<b>rmonSetComport()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised and Idle State						
<b>Synopsis</b>	rmonRet __stdcall rmonSetComport(HRMONCON hCon, const char* LocalPort, const char* RemotePort)							
<b>Description</b>	<p>The connection needs to know which serial ports on the PC is going to be used for communication, both for direct cable connection, and for connection through a modem. These ports are configured using this call. If one of the ports is not to be used, simply set it to "COM0".</p> <p>To configure an USB cable connection, use the port names "USB1" through "USB8".</p>							
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection.</td> </tr> <tr> <td><b>LocalPort</b></td> <td>Name of the COM port to be used for direct cable connection.</td> </tr> <tr> <td><b>RemotePort</b></td> <td>Name of the COM port to be used for remote connection through a modem.</td> </tr> </table>		<b>hCon</b>	Handle to connection.	<b>LocalPort</b>	Name of the COM port to be used for direct cable connection.	<b>RemotePort</b>	Name of the COM port to be used for remote connection through a modem.
<b>hCon</b>	Handle to connection.							
<b>LocalPort</b>	Name of the COM port to be used for direct cable connection.							
<b>RemotePort</b>	Name of the COM port to be used for remote connection through a modem.							
<b>Returns</b>	rmonOK, rmonConnection, rmonIllegalHandle							

<b>rmonSetModemInit()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised and Idle State
<b>Synopsis</b>	rmonRet __stdcall rmonSetModemInit(HRMONCON hCon, const char* atcmd)	
<b>Description</b>	Specifies initialisation string to modem for usage in remote connection.	
<b>Input</b>	<b>hCon</b>	Handle to the connection.
	<b>Atcmd</b>	Initialisation string for modem
<b>Returns</b>	rmonOK, rmonConnection, rmonIllegalHandle	

<b>rmonSetRemoteBaudrate()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Not Initialised and Idle State
<b>Synopsis</b>	rmonRet __stdcall rmonSetRemoteBaudrate(HRMONCON hCon, int baud)	
<b>Description</b>	This function sets the baud rate that will be used when communication is established to a remote device through a modem. This is the baud rate used between the PC and the Modem, and has nothing to do with the speed being used on the link between the modem and RTCU device (which can be influenced by setting the appropriate settings in the rmonSetModemInit()).	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Baud</b>	Baud rate to be used. If baud is set to 0 a default value of 57600 baud is used. If allowed by hardware the baud rate in principle can be set to anything Commonly used baud rates are: 9600, 19200, 38400, 57600 and 115200 Other protocol parameters are: No parity, 8 data bits and 1 stop bit.
<b>Returns</b>	rmonOK, rmonConnection, rmonIllegalHandle	

<b>rmonConnect()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Idle
<b>Synopsis</b>	rmonRet __stdcall rmonConnect(HRMONCON hCon, const char* phonenumber)	
<b>Description</b>	This function is used to establish a connection to a RTCU device. The connection can be either over cable, through a modem, or through the RTCU Communication Hub. If the remote RTCU is to be contacted through a modem, simply use the telephone number of the SIM card in the RTCU, if connection is through the RTCU Communication Hub, the devices serial number (nodeid) is to be used, prefixed with a "@" character! To establish a connection over cable, leave the phone number empty.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>onenumber</b>	The phone number of the SIM card in the remote RTCU if a connection is through modem, or the serial number of the RTCU (prefixed with "@") if connection is through RCH.
<b>Returns</b>	rmonOK, rmonIllegalHandle, rmonConnection, rmonDenied, rmonComError, rmonError	

<b>rmonDisconnect()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> Remotely Connected State		
<b>Synopsis</b>	rmonRet __stdcall rmonDisconnect(HRMONCON hCon)			
<b>Description</b>	Disconnect a connection to a RTCU device.			
<b>Input</b>	<table border="1"><tr><td><b>hCon</b></td><td>Handle to connection</td></tr></table>	<b>hCon</b>	Handle to connection	
<b>hCon</b>	Handle to connection			
<b>Returns</b>	rmonOK, rmonIllegalHandle, rmonError			

<b>rmonConnected()</b>		<b>RTCU architecture:</b> n/a <b>Called in:</b> After Initialised State															
<b>Synopsis</b>	rmonRet __stdcall rmonConnected(HRMONCON hCon)																
<b>Description</b>	rmonConnected() returns the type of connection that is currently (if any) established with the RTCU device.																
<b>Input</b>	<table border="1"><tr><td><b>hCon</b></td><td>Handle to connection</td></tr></table>	<b>hCon</b>	Handle to connection														
<b>hCon</b>	Handle to connection																
<b>Returns</b>	Type of connection:																
	<table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td><b>RMONCON_NONE</b></td> <td>0</td> <td>Currently not connected</td> </tr> <tr> <td><b>RMONCON_LOCAL</b></td> <td>1</td> <td>Connected using cable</td> </tr> <tr> <td><b>RMONCON_REMOTE</b></td> <td>2</td> <td>Connected through a modem</td> </tr> <tr> <td><b>RMONCON_GW</b></td> <td>3</td> <td>Connected through the RCH</td> </tr> </tbody> </table>	Symbolic name	Value	Description	<b>RMONCON_NONE</b>	0	Currently not connected	<b>RMONCON_LOCAL</b>	1	Connected using cable	<b>RMONCON_REMOTE</b>	2	Connected through a modem	<b>RMONCON_GW</b>	3	Connected through the RCH	
Symbolic name	Value	Description															
<b>RMONCON_NONE</b>	0	Currently not connected															
<b>RMONCON_LOCAL</b>	1	Connected using cable															
<b>RMONCON_REMOTE</b>	2	Connected through a modem															
<b>RMONCON_GW</b>	3	Connected through the RCH															

<b>rmonAuthenticate()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Not Authenticated State				
<b>Synopsis</b>	rmonRet __stdcall rmonAuthenticate (HRMONCON hCon, const char password[21])					
<b>Description</b>	<p>if there is established a (new) connection with a RTCU device, either local or remote, the first thing to do, is for the application to authenticate itself for the RTCU device. This is done using this function, and must be done, BEFORE any other communication with the device can take place.</p> <p>If there is no password set in the RTCU device, this function must still be called, just with an empty string "" as the password.</p>					
<b>Input</b>	<table border="1"><tr><td><b>hCon</b></td><td>Handle to connection</td></tr><tr><td><b>password</b></td><td>Password, zero terminated ASCII string</td></tr></table>	<b>hCon</b>	Handle to connection	<b>password</b>	Password, zero terminated ASCII string	
<b>hCon</b>	Handle to connection					
<b>password</b>	Password, zero terminated ASCII string					
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonIllegalTarget, rmonTargetError, rmonDenied					

<b>rmonEnableLargeData()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Not Initialised and Idle State
<b>Synopsis</b>	rmonRet __stdcall rmonEnableLargeData(HRMONCON hCon, unsigned char enable)	
<b>Description</b>	This function enables the support for larger data frames on medias that supports them. This allows for transferring more data for each call to some functions, including rmonLogReadExt and rmonLogReadByTag.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>enable</b>	0=disable, 1=enable
<b>Returns</b>	rmonOK, rmonIllegalHandle	

## Program/Firmware upload

The following functions are used for uploading new applications, voice messages and firmware to a RTCU device:

All functions report their progress, by calling an optional call-back function, defined as follows:

```
typedef int (__stdcall *rmoncbprogress)(void* uptr,int percent);
```

Please note that if the call-back function returns a value different from 0 (zero) the functions will cancel.

<b>rmonFirmwareUpload()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Locally Connected State								
<b>Synopsis</b>	rmonRet __stdcall rmonFirmwareUpload(HRMONCON hCon, char *FirmwareFilename, rmoncbprogress cbfunc, void *uptr);									
<b>Description</b>	Upload firmware to RTCU. The function takes a .BIN firmware file, and transfers it to an RTCU device. The function will halt execution in the device, upload the new firmware file, and after the transfer, it will reset the device. Note that to upload a new firmware to an RTCU device when the connection to it is via the RCH, you need to use background update (see rmonFirmwareStartUpload).									
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>FirmwareFilename</b></td> <td>Firmware file</td> </tr> <tr> <td><b>cbfunc</b></td> <td>Call-back function for progress</td> </tr> <tr> <td><b>uptr</b></td> <td>User data that will be passed to call-back function when called</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>FirmwareFilename</b>	Firmware file	<b>cbfunc</b>	Call-back function for progress	<b>uptr</b>	User data that will be passed to call-back function when called
<b>hCon</b>	Handle to connection									
<b>FirmwareFilename</b>	Firmware file									
<b>cbfunc</b>	Call-back function for progress									
<b>uptr</b>	User data that will be passed to call-back function when called									
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonIllegalTarget, rmonNotGateway, rmonNoMonitorMode, rmonErrorReset, rmonCancelled, rmonFileNotFound, rmonIllegalFile, rmonOldFormat, rmonNotModem, rmonDowngrade									

<b>rmonFirmwareStartUpload()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State										
<b>Synopsis</b>	rmonRet __stdcall rmonFirmwareStartUpload(HRMONCON hCon, const char* Filename, struct rmonBGReport* Report, rmoncbprogress cbfunc, void* uptr );											
<b>Description</b>	<p>Upload firmware to RTCU.</p> <p>Function takes a .BIN firmware file, and transfers it to a RTCU device. Unlike rmonFirmwareUpload the function will not halt execution in the device, but start to upload the firmware in the background. The upload started with this function supports resume if the upload is interrupted. If the upload is interrupted the Report structure will contain the information needed to resume the upload using rmonFirmwareResumeUpload. The newly uploaded firmware is used after the device has been reset. Note that part of the voice memory in the X32 device is used for the update, and any voice data must be uploaded again. Use the function rmonVoiceMessagesAbove64K to determine if the use of this function will overwrite any voice data.</p>											
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>Filename</b></td> <td>Zero terminated string with the firmware filename</td> </tr> <tr> <td><b>Report</b></td> <td>A structure containing progress status (see definition below)</td> </tr> <tr> <td><b>cbfunc</b></td> <td>Call-back function for progress</td> </tr> <tr> <td><b>uptr</b></td> <td>User data that will be passed to call-back function when called</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>Filename</b>	Zero terminated string with the firmware filename	<b>Report</b>	A structure containing progress status (see definition below)	<b>cbfunc</b>	Call-back function for progress	<b>uptr</b>	User data that will be passed to call-back function when called
<b>hCon</b>	Handle to connection											
<b>Filename</b>	Zero terminated string with the firmware filename											
<b>Report</b>	A structure containing progress status (see definition below)											
<b>cbfunc</b>	Call-back function for progress											
<b>uptr</b>	User data that will be passed to call-back function when called											
<b>Returns</b>	<p>rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonCancelled, rmonNoBackground, rmonFileNotFound, rmonOldFormat, rmonIllegalFile, rmonIllegalTarget, rmonInterrupted, rmonDowngrade</p> <pre> struct rmonBGReport {     unsigned long  extSeg;      // External segment written     unsigned long  intSeg;      // Internal segment written     unsigned long  headSeg;     // Header segment written }; </pre>											

<b>rmonFirmwareResumeUpload()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State										
<b>Synopsis</b>	rmonRet __stdcall rmonFirmwareResumeUpload(HRMONCON hCon, const char* Filename, struct rmonBGReport* Report, rmoncbprogress cbfunc, void* uptr);											
<b>Description</b>	<p>Upload firmware to RTCU.</p> <p>Function takes a report containing a .BIN firmware file and progress status, and resumes where the upload was interrupted. If the upload is interrupted the Report structure will contain the information needed to resume the upload.</p> <p>Note that this function cannot be used to start a new upload, only complete an interrupted upload.</p> <p>Note that part of the voice memory in the X32 device is used for the update and voice data must be uploaded again.</p>											
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>Filename</b></td> <td>Zero terminated string with the firmware filename</td> </tr> <tr> <td><b>Report</b></td> <td>A structure containing progress status (see definition below)</td> </tr> <tr> <td><b>cbfunc</b></td> <td>Call-back function for progress</td> </tr> <tr> <td><b>uptr</b></td> <td>User data that will be passed to call-back function when called</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>Filename</b>	Zero terminated string with the firmware filename	<b>Report</b>	A structure containing progress status (see definition below)	<b>cbfunc</b>	Call-back function for progress	<b>uptr</b>	User data that will be passed to call-back function when called
<b>hCon</b>	Handle to connection											
<b>Filename</b>	Zero terminated string with the firmware filename											
<b>Report</b>	A structure containing progress status (see definition below)											
<b>cbfunc</b>	Call-back function for progress											
<b>uptr</b>	User data that will be passed to call-back function when called											
<b>Returns</b>	<p>rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonCancelled, rmonNoBackground, rmonFileNotFound, rmonOldFormat, rmonIllegalFile, rmonIllegalTarget, rmonInterrupted</p> <pre> struct rmonBGReport{     unsigned long  extSeg;      // External segment written     unsigned long  intSeg;     // Internal segment written     unsigned long  headSeg;    // Header segment written }; </pre>											

<b>rmonApplicationUpload()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State								
<b>Synopsis</b>	rmonRet __stdcall rmonApplicationUpload(HRMONCON hCon, char *Filename, rmoncbprogress cbfunc, void *uptr);									
<b>Description</b>	<p>Uploads application to RTCU</p> <p>Function takes a .VSX, a .PSX or a .RPC file, and transfers it to a RTCU.</p> <p>Please note that the execution of the VPL program in the RTCU device <b>must</b> be halted with rmonHalt(), <b>before</b> calling this function ! The RTCU will have to be reset after the transfer, to start the new application.</p>									
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>Filename</b></td> <td>Zero terminated string with the filename of the application</td> </tr> <tr> <td><b>cbfunc</b></td> <td>Call back function for progress</td> </tr> <tr> <td><b>uptr</b></td> <td>User data that will be passed to call back function when called</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>Filename</b>	Zero terminated string with the filename of the application	<b>cbfunc</b>	Call back function for progress	<b>uptr</b>	User data that will be passed to call back function when called
<b>hCon</b>	Handle to connection									
<b>Filename</b>	Zero terminated string with the filename of the application									
<b>cbfunc</b>	Call back function for progress									
<b>uptr</b>	User data that will be passed to call back function when called									
<b>Returns</b>	<p>rmonOK, rmonComError, rmonIllegalHandle, rmonErrorHalt, rmonFileNotFound, rmonIllegalFile, rmonNotProgrammable, rmonCancelled, rmonTargetError, rmonIllegalTarget, rmonImageTooLarge, rmonImageSupport</p>									

<b>rmonApplicationStartUpload()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State										
<b>Synopsis</b>	rmonRet __stdcall rmonApplicationStartUpload(HRMONCON hCon, const char* Filename, struct rmonBGReport* Report, rmoncbprogress cbfunc, void *uptr);											
<b>Description</b>	<p>Uploads application to RTCU</p> <p>Function takes a Report containing a .VSX, a .PSX or a .RPC file, and transfers the file to a RTCU. The upload will be performed in the background without interfering with the running application. The upload started with this function supports resume if the upload is interrupted. If the upload is interrupted the Report structure will contain the information needed to resume the upload using rmonApplicationResumeUpload.</p> <p>The newly uploaded application is used after the device has been reset.</p> <p>Note that the voice memory in the X32 device is used, and any voice data must be uploaded again. Use the function rmonVoiceMessagesAbove64K to determine if the use of this function will overwrite any voice data.</p>											
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>Filename</b></td> <td>Zero terminated string with the filename of the application</td> </tr> <tr> <td><b>Report</b></td> <td>A structure containing progress status (see definition below)</td> </tr> <tr> <td><b>cbfunc</b></td> <td>Call back function for progress</td> </tr> <tr> <td><b>uptr</b></td> <td>User data that will be passed to call back function when called</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>Filename</b>	Zero terminated string with the filename of the application	<b>Report</b>	A structure containing progress status (see definition below)	<b>cbfunc</b>	Call back function for progress	<b>uptr</b>	User data that will be passed to call back function when called
<b>hCon</b>	Handle to connection											
<b>Filename</b>	Zero terminated string with the filename of the application											
<b>Report</b>	A structure containing progress status (see definition below)											
<b>cbfunc</b>	Call back function for progress											
<b>uptr</b>	User data that will be passed to call back function when called											
<b>Returns</b>	<p>rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonIllegalTarget, rmonCancelled, rmonNoBackground, rmonFileNotFound, rmonIllegalFile, rmonNotProgrammable, rmonInterrupted, rmonImageTooLarge, rmonImageSupport</p> <pre> struct rmonBGReport {     unsigned long  extSeg;      // External segment written     unsigned long  intSeg;     // Internal segment written     unsigned long  headSeg;    // Header segment written }; </pre>											

<b>rmonApplicationResumeUpload()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State										
<b>Synopsis</b>	rmonRet __stdcall rmonApplicationResumeUpload(HRMONCON hCon, const char* Filename, struct rmonBGReport* Report, rmoncbprogress cbfunc, void *uptr);											
<b>Description</b>	<p>Uploads application to RTCU</p> <p>Function takes a report containing a .VSX, a .PSX or a .RPC file and progress status, and resumes where the upload was interrupted. If the upload is interrupted the Report structure will contain the information needed to resume the upload.</p> <p>Note that this function cannot be used to start a new upload, only complete an interrupted upload.</p> <p>Note that the voice memory in the X32 device is used, and any voice data must be uploaded again.</p>											
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>Filename</b></td> <td>Zero terminated string with the filename of the application</td> </tr> <tr> <td><b>Report</b></td> <td>A structure containing progress status (see definition below)</td> </tr> <tr> <td><b>cbfunc</b></td> <td>Call back function for progress</td> </tr> <tr> <td><b>uptr</b></td> <td>User data that will be passed to call back function when called</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>Filename</b>	Zero terminated string with the filename of the application	<b>Report</b>	A structure containing progress status (see definition below)	<b>cbfunc</b>	Call back function for progress	<b>uptr</b>	User data that will be passed to call back function when called
<b>hCon</b>	Handle to connection											
<b>Filename</b>	Zero terminated string with the filename of the application											
<b>Report</b>	A structure containing progress status (see definition below)											
<b>cbfunc</b>	Call back function for progress											
<b>uptr</b>	User data that will be passed to call back function when called											
<b>Returns</b>	<p>rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonIllegalTarget, rmonCancelled, rmonNoBackground, rmonFileNotFound, rmonIllegalFile, rmonNotProgrammable, rmonInterrupted, rmonImageTooLarge, rmonImageSupport</p> <pre> struct rmonBGReport {     unsigned long  extSeg;      // External segment written     unsigned long  intSeg;     // Internal segment written     unsigned long  headSeg;    // Header segment written }; </pre>											

<b>rmonVoiceUpload()</b>		<b>RTC architecture:</b> X32 & NX32 <b>Called in:</b> Connected State								
<b>Synopsis</b>	rmonRet __stdcall rmonVoiceUpload(HRMONCON hCon, char *ProjectFilename, rmoncbprogress cbfunc, void *uptr);									
<b>Description</b>	<p>Upload Voice messages to RTCU.</p> <p>Function transfers all voice messages associated with a RTCU project.</p> <p>It is important that the relative directory structure for the project is kept the same as when the project was built in the RTCU M2M Studio environment, otherwise the function will have trouble locating the voice message files.</p> <p>Please note that the execution of the VPL program in the RTCU device must be halted with rmonHalt(), before calling this function ! The RTCU will have to be reset after the transfer for the new voice messages to take effect.</p>									
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>ProjectFilename</b></td> <td>Project filename</td> </tr> <tr> <td><b>cbfunc</b></td> <td>Call back function for progress</td> </tr> <tr> <td><b>uptr</b></td> <td>User data that will be passed to call back function when called</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>ProjectFilename</b>	Project filename	<b>cbfunc</b>	Call back function for progress	<b>uptr</b>	User data that will be passed to call back function when called
<b>hCon</b>	Handle to connection									
<b>ProjectFilename</b>	Project filename									
<b>cbfunc</b>	Call back function for progress									
<b>uptr</b>	User data that will be passed to call back function when called									
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonCancelled, rmonFileNotFound, rmonIllegalFile, rmonMemoryConfig, rmonImageTooLarge									

<b>rmonNumOfVoiceMessages()</b>		<b>RTC architecture:</b> n/a <b>Called in:</b> All states				
<b>Synopsis</b>	rmonRet __stdcall rmonNumOfVoiceMessages(char *ProjectFilename, int *NumFiles);					
<b>Description</b>	Determine how many voice messages is included in a project. This is useful for determining if the rmonVoiceUpload() function has to be called when uploading a complete project to a RTCU device.					
<b>Input</b>	<table border="1"> <tr> <td><b>ProjectFilename</b></td> <td><b>Project filename</b></td> </tr> <tr> <td><b>NumFile</b></td> <td>Number of voice file in PRJ file</td> </tr> </table>	<b>ProjectFilename</b>	<b>Project filename</b>	<b>NumFile</b>	Number of voice file in PRJ file	
<b>ProjectFilename</b>	<b>Project filename</b>					
<b>NumFile</b>	Number of voice file in PRJ file					
<b>Returns</b>	rmonOK, rmonFileNotFound, rmonIllegalFile					

<b>rmonCheckTransfer()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State																		
<b>Synopsis</b>	rmonRet __stdcall rmonCheckTransfer(HRMONCON hCon, int type, const char* filename, int* report);																			
<b>Description</b>	Determine if there are any special considerations for performing a transfer.																			
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td><b>Handle to connection.</b></td> </tr> <tr> <td><b>Type</b></td> <td>The type of transfer to check, see below.</td> </tr> <tr> <td><b>filename</b></td> <td>The name of the file to transfer.</td> </tr> </table>	<b>hCon</b>	<b>Handle to connection.</b>	<b>Type</b>	The type of transfer to check, see below.	<b>filename</b>	The name of the file to transfer.													
<b>hCon</b>	<b>Handle to connection.</b>																			
<b>Type</b>	The type of transfer to check, see below.																			
<b>filename</b>	The name of the file to transfer.																			
<b>Output</b>	<table border="1"> <tr> <td><b>report</b></td> <td>Report on special conditions, see below.</td> </tr> </table>	<b>report</b>	Report on special conditions, see below.																	
<b>report</b>	Report on special conditions, see below.																			
<b>Returns</b>	rmonOK, rmonFileNotFound, rmonIllegalFile																			
	Type of transfer:																			
	<table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>RMONCT_APP_DIRECT</td> <td>1</td> <td>Direct application transfer.</td> </tr> <tr> <td>RMONCT_APP_BACKGROUND</td> <td>2</td> <td>Background application transfer.</td> </tr> <tr> <td>RMONCT_FW_DIRECT</td> <td>3</td> <td>Direct firmware transfer.</td> </tr> <tr> <td>RMONCT_FW_BACKGROUND</td> <td>4</td> <td>Background transfer.</td> </tr> <tr> <td>RMONCT_VOICE</td> <td>5</td> <td>Voice transfer.</td> </tr> </tbody> </table>	Symbolic name	Value	Description	RMONCT_APP_DIRECT	1	Direct application transfer.	RMONCT_APP_BACKGROUND	2	Background application transfer.	RMONCT_FW_DIRECT	3	Direct firmware transfer.	RMONCT_FW_BACKGROUND	4	Background transfer.	RMONCT_VOICE	5	Voice transfer.	
Symbolic name	Value	Description																		
RMONCT_APP_DIRECT	1	Direct application transfer.																		
RMONCT_APP_BACKGROUND	2	Background application transfer.																		
RMONCT_FW_DIRECT	3	Direct firmware transfer.																		
RMONCT_FW_BACKGROUND	4	Background transfer.																		
RMONCT_VOICE	5	Voice transfer.																		
	Transfer report:																			
	<table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>RMONCT_REP_VOICE</td> <td>1</td> <td>Voice in device will be erased.</td> </tr> <tr> <td>RMONCT_REP_FW300</td> <td>2</td> <td>Firmware version 3.00 is required.</td> </tr> <tr> <td>RMONCT_REP_DENIED</td> <td>3</td> <td>Voice transfer is not possible.</td> </tr> </tbody> </table>	Symbolic name	Value	Description	RMONCT_REP_VOICE	1	Voice in device will be erased.	RMONCT_REP_FW300	2	Firmware version 3.00 is required.	RMONCT_REP_DENIED	3	Voice transfer is not possible.							
Symbolic name	Value	Description																		
RMONCT_REP_VOICE	1	Voice in device will be erased.																		
RMONCT_REP_FW300	2	Firmware version 3.00 is required.																		
RMONCT_REP_DENIED	3	Voice transfer is not possible.																		

<b>rmonApplicationFilename()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State		
<b>Synopsis</b>	rmonRet __stdcall rmonApplicationFilename( const char* ProjectFilename, char* Filename);			
<b>Description</b>	Provided with the name of the project file, this function determines the name of the corresponding application file.			
<b>Input</b>	<table border="1"> <tr> <td><b>ProjectFilename</b></td> <td>ASCIIZ string with the name of the project file with the file extension .PRJ</td> </tr> </table>		<b>ProjectFilename</b>	ASCIIZ string with the name of the project file with the file extension .PRJ
<b>ProjectFilename</b>	ASCIIZ string with the name of the project file with the file extension .PRJ			
<b>Output</b>	<table border="1"> <tr> <td><b>Filename</b></td> <td>Name of application file generated from project. Must be large enough to store a filename with the same length as ProjectFilename.</td> </tr> </table>		<b>Filename</b>	Name of application file generated from project. Must be large enough to store a filename with the same length as ProjectFilename.
<b>Filename</b>	Name of application file generated from project. Must be large enough to store a filename with the same length as ProjectFilename.			
<b>Returns</b>	rmonOK, rmonFileNotFound, rmonIllegalFile			

## Manipulation of Persistent memory

The Persistent memory of the RTCU device, can be manipulated with this set of functions. The FLASH based Persistent memory in the RTCU devices, is accessible from VPL using the functions SaveData/LoadData, SaveString/LoadString. The FRAM based memory, is accessible with the functions SaveDataF / LoadDataF, SaveStringF / LoadStringF.

<b>rmonPersistentRead()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State	
<b>Synopsis</b>	rmonRet __stdcall rmonPersistentRead( HRMONCON hCon, int first, int last, int type, char* data, rmoncbprogress pfunc, void* uptr, int reserved);		
<b>Description</b>	Reads a range of entries from Persistent memory with bounds check.		
<b>Input</b>	<b>hCon</b>	<b>Handle to connection</b>	
	<b>first</b>	1 based index of first entry to read	
	<b>last</b>	1 based index of last entry to read	
	<b>type</b>	The type of persistent memory to read from, see below	
	<b>pfunc</b>	Pointer to progress callback function	
	<b>uptr</b>	Pointer to user argument used when reporting progress.	
	<b>reserved</b>	Reserved for future use. Must be set to zero	
<b>Output</b>	<b>data</b>	Buffer to store the data in. The buffer is handled as a packed array of the structure rmonPersistEntry, see below.	
	<b>Returns</b>		
rmonOK, rmonComError, rmonIllegalHandle, rmonError, rmonCancelled, rmonIllegalAccess			
type:			
<b>Symbolic name</b>		<b>Value</b>	<b>Description</b>
<b>RMON_TYPE_FRAM</b>		1	FRAM
<b>RMON_TYPE_FLASH</b>		2	FLASH
<b>RMON_TYPE_XFLASH</b>		3	Exterded FLASH
<pre> struct rmonPersistEntry {     unsigned char type;           // The type of entry, see below     short length;                // The number of bytes in the entry     unsigned char data[255];     // The contents of the persistent entry }; </pre>			
type:			
<b>Symbolic name</b>		<b>Value</b>	<b>Description</b>
<b>RMON_PERSIST_TEXT</b>		1	Text entry
<b>RMON_PERSIST_BINARY</b>		2	Binary entry

<b>rmonPersistentWrite()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State												
<b>Synopsis</b>	rmonRet __stdcall rmonPersistentRead( HRMONCON hCon, int first, int last, int type, char* data, rmoncbprogress pfunc, void* uptr, int reserved );													
<b>Description</b>	Writes a range of entries from Persistent memory with bounds check.													
<b>Input</b>	<b>hCon</b>	<b>Handle to connection</b>												
	<b>first</b>	1 based index of first entry to read												
	<b>last</b>	1 based index of last entry to read												
	<b>type</b>	The type of persistent memory to write to, see below												
	<b>data</b>	Buffer with entries to write. The buffer is handled as a packed array of the structure rmonPersistEntry, see below.												
	<b>pfunc</b>	Pointer to progress callback function												
	<b>uptr</b>	Pointer to user argument used when reporting progress.												
	<b>reserved</b>	Reserved for future use. Must be set to zero												
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonError, rmonIllegalAccess													
	type:													
	<table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td><b>RMON_TYPE_FRAM</b></td> <td>1</td> <td>FRAM</td> </tr> <tr> <td><b>RMON_TYPE_FLASH</b></td> <td>2</td> <td>FLASH</td> </tr> <tr> <td><b>RMON_TYPE_XFLASH</b></td> <td>3</td> <td>Exterded FLASH</td> </tr> </tbody> </table>		Symbolic name	Value	Description	<b>RMON_TYPE_FRAM</b>	1	FRAM	<b>RMON_TYPE_FLASH</b>	2	FLASH	<b>RMON_TYPE_XFLASH</b>	3	Exterded FLASH
Symbolic name	Value	Description												
<b>RMON_TYPE_FRAM</b>	1	FRAM												
<b>RMON_TYPE_FLASH</b>	2	FLASH												
<b>RMON_TYPE_XFLASH</b>	3	Exterded FLASH												
	<pre> struct rmonPersistEntry {     unsigned char type;           // The type of entry, see below     short length;                // The number of bytes in the entry     unsigned char data[255];     // The contents of the persistent entry }; </pre>													
	type:													
	<table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td><b>RMON_PERSIST_TEXT</b></td> <td>1</td> <td>Text entry</td> </tr> <tr> <td><b>RMON_PERSIST_BINARY</b></td> <td>2</td> <td>Binary entry</td> </tr> </tbody> </table>		Symbolic name	Value	Description	<b>RMON_PERSIST_TEXT</b>	1	Text entry	<b>RMON_PERSIST_BINARY</b>	2	Binary entry			
Symbolic name	Value	Description												
<b>RMON_PERSIST_TEXT</b>	1	Text entry												
<b>RMON_PERSIST_BINARY</b>	2	Binary entry												

<b>rmonReadPersistentFRAM()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonReadPersistentFRAM(HRMONCON hCon, int entry, char *data, int *length, int binary);	
<b>Description</b>	<p>Read FRAM based persistent entry.</p> <p>This function reads a specific entry from FRAM based Persistent memory in the RTCU. When called, you must specify what type of data you are expecting to read, if the specified type of data is not present, the function returns rmonNoData, otherwise the data and length are returned.</p> <p>Data read with this function, can be stored from VPL with the functions SaveStringF() and SaveDataF()</p>	
<b>Input</b>	<b>hCon</b>	<b>Handle to connection</b>
	<b>entry</b>	Entry number, from 1 to 20 on X32 and from 1 to 100 on NX32 and NX32L.
	<b>binary</b>	Set to 1 if expecting binary data, 0 if string expected
<b>Output</b>	<b>data</b>	<b>Buffer for data (must be large enough!)</b>
	<b>length</b>	The number of bytes read from the entry
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData, rmonError	

<b>rmonWritePersistentFRAM()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonWritePersistentFRAM(HRMONCON hCon, int entry, char *data, int length, int binary);	
<b>Description</b>	<p>Write to FRAM based persistent entry.</p> <p>This function writes either binary data or a string to a specific entry in the FRAM based persistent memory in the RTCU. When called, you must specify what type of data you are storing.</p> <p>Data stored with this function, can be read from VPL with the functions LoadStringF() and LoadDataF().</p>	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>entry</b>	Entry number, from 1 to 20 on X32 and from 1 to 100 on NX32 and NX32L.
	<b>data</b>	The data to store
	<b>length</b>	Length of data
	<b>binary</b>	Set to 1 if storing binary data, 0 if string
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonError	

<b>rmonReadPersistentFLASH()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State						
<b>Synopsis</b>	rmonRet __stdcall rmonReadPersistentFLASH(HRMONCON hCon, int entry, char *data, int *length, int binary);							
<b>Description</b>	<p>Read FLASH based persistent entry</p> <p>This function reads a specific entry from FLASH based persistent memory in the RTCU. When called, you must specify what type of data you are expecting to read, if the specified type of data is not present, the function returns rmonNoData, otherwise the data and length are returned.</p> <p>Data read with this function can be stored from VPL with the functions SaveString() and SaveData().</p>							
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>entry</b></td> <td>Entry number, from 1 to 192</td> </tr> <tr> <td><b>binary</b></td> <td>Set to 1 if expecting binary data, 0 if string expected</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>entry</b>	Entry number, from 1 to 192	<b>binary</b>	Set to 1 if expecting binary data, 0 if string expected
<b>hCon</b>	Handle to connection							
<b>entry</b>	Entry number, from 1 to 192							
<b>binary</b>	Set to 1 if expecting binary data, 0 if string expected							
<b>Output</b>	<table border="1"> <tr> <td><b>data</b></td> <td>Buffer for data (must be large enough!)</td> </tr> <tr> <td><b>length</b></td> <td>The number of bytes read from the entry</td> </tr> </table>		<b>data</b>	Buffer for data (must be large enough!)	<b>length</b>	The number of bytes read from the entry		
<b>data</b>	Buffer for data (must be large enough!)							
<b>length</b>	The number of bytes read from the entry							
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData, rmonError							

<b>rmonWritePersistentFLASH()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State										
<b>Synopsis</b>	rmonRet __stdcall rmonWritePersistentFLASH(HRMONCON hCon, int entry, char *data, int length, int binary);											
<b>Description</b>	<p>Write to FLASH based persistent entry.</p> <p>This function writes either binary data or a string to a specific entry in the FLASH based persistent memory in the RTCU. When called, you must specify what type of data you are storing.</p> <p>Data stored with this function, can be read from VPL with the functions LoadString() and LoadData().</p>											
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>entry</b></td> <td>Entry number, from 1 to 192</td> </tr> <tr> <td><b>data</b></td> <td>The data to store</td> </tr> <tr> <td><b>length</b></td> <td>Length of data</td> </tr> <tr> <td><b>binary</b></td> <td>Set to 1 if storing binary data, 0 if string</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>entry</b>	Entry number, from 1 to 192	<b>data</b>	The data to store	<b>length</b>	Length of data	<b>binary</b>	Set to 1 if storing binary data, 0 if string
<b>hCon</b>	Handle to connection											
<b>entry</b>	Entry number, from 1 to 192											
<b>data</b>	The data to store											
<b>length</b>	Length of data											
<b>binary</b>	Set to 1 if storing binary data, 0 if string											
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonError											

<b>rmonGetXFLASHSize()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonGetXFLASHSize(HRMONCON hCon, int *size);	
<b>Description</b>	Get the number of entries in extended FLASH. This function is identical to the VPL function GetFlashXSize().	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>size</b>	Number of entries in extended flash.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonReadPersistentXFLASH()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonReadPersistentXFLASH(HRMONCON hCon, int entry, char *data, int *length, int binary);	
<b>Description</b>	Read extended FLASH based persistent entry  This function reads a specific entry from extended FLASH based persistent memory in the RTCU. When called, you must specify what type of data you are expecting to read, if the specified type of data is not present, the function returns rmonNoData, otherwise the data and length are returned. Data read with this function can be stored from VPL with the functions SaveStringX() and SaveDataX().	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>entry</b>	Entry number, from 1 to Size (Determined with the function rmonGetXFLASHSize())
	<b>binary</b>	Set to 1 if expecting binary data, 0 if string expected
<b>Output</b>	<b>data</b>	Buffer for data (must be large enough!)
	<b>length</b>	The number of bytes read from the entry
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError. rmonError, rmonNoData	

<b>rmonWritePersistentXFLASH()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State										
<b>Synopsis</b>	rmonRet __stdcall rmonWritePersistentXFLASH(HRMONCON hCon, int entry, char *data, int length, int binary);											
<b>Description</b>	<p>Write to extended FLASH based persistent entry.</p> <p>This function writes either binary data or a string to a specific entry in the extended FLASH based persistent memory in the RTCU. When called, you must specify what type of data you are storing.</p> <p>Data stored with this function, can be read from VPL with the functions LoadStringX() and LoadDataX().</p>											
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>entry</b></td> <td>Entry number, from 1 to Size (Determined with the function rmonGetXFLASHSize())</td> </tr> <tr> <td><b>data</b></td> <td>The data to store</td> </tr> <tr> <td><b>length</b></td> <td>Length of data</td> </tr> <tr> <td><b>binary</b></td> <td>Set to 1 if storing binary data, 0 if string</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>entry</b>	Entry number, from 1 to Size (Determined with the function rmonGetXFLASHSize())	<b>data</b>	The data to store	<b>length</b>	Length of data	<b>binary</b>	Set to 1 if storing binary data, 0 if string
<b>hCon</b>	Handle to connection											
<b>entry</b>	Entry number, from 1 to Size (Determined with the function rmonGetXFLASHSize())											
<b>data</b>	The data to store											
<b>length</b>	Length of data											
<b>binary</b>	Set to 1 if storing binary data, 0 if string											
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError. rmonError											

## Datalogger

The built-in datalogger of the RTCU device can be manipulated with this set of functions. The log can be read, searched and cleared etc. For a more detailed description of the datalogger in the RTCU devices, please refer to the online help for the RTCU M2M Studio.

The read and write pointer is NOT shared with the VPL application

<b>rmonLogFirst()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonLogFirst(HRMONCON hCon)	
<b>Description</b>	Moves the current read pointer to the first (oldest) entry in the datalogger	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonLogLast()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonLogLast(HRMONCON hCon)	
<b>Description</b>	Moves the current read pointer to the last (newest) entry in the datalogger.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonLogReadExt()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State																											
<b>Synopsis</b>	rmonRet __stdcall rmonLogReadExt(HRMONCON hCon, int operation, int* values_per_rec, int* entries_in_buffer, char* buffer);																												
<b>Description</b>	This function reads up to 146 log entries from the datalogger each time it is called. When it is called, the current read pointer is either incremented or decremented, according to the operation parameter.																												
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>operation</b></td> <td>RMONLOGGET_NEXT or RMONLOGGET_PREV</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>operation</b>	RMONLOGGET_NEXT or RMONLOGGET_PREV																							
<b>hCon</b>	Handle to connection																												
<b>operation</b>	RMONLOGGET_NEXT or RMONLOGGET_PREV																												
<b>Output</b>	<table border="1"> <tr> <td><b>values_per_rec</b></td> <td>Number of values in each log entry, maximum 8</td> </tr> <tr> <td><b>entries_in_buffer</b></td> <td>Number of entries in the buffer, maximum 146</td> </tr> <tr> <td><b>buffer</b></td> <td>Buffer containing the read entries. The entries are placed directly after each other, and the format of each entry can be seen below. The required size of the buffer depends on the type of connection: Cable (standard data size): 240 bytes. Remote (standard data size): 480 bytes. When large data is enabled (see rmonEnableLargeData()): Cable (large data size): 1024 bytes.</td> </tr> </table>		<b>values_per_rec</b>	Number of values in each log entry, maximum 8	<b>entries_in_buffer</b>	Number of entries in the buffer, maximum 146	<b>buffer</b>	Buffer containing the read entries. The entries are placed directly after each other, and the format of each entry can be seen below. The required size of the buffer depends on the type of connection: Cable (standard data size): 240 bytes. Remote (standard data size): 480 bytes. When large data is enabled (see rmonEnableLargeData()): Cable (large data size): 1024 bytes.																					
<b>values_per_rec</b>	Number of values in each log entry, maximum 8																												
<b>entries_in_buffer</b>	Number of entries in the buffer, maximum 146																												
<b>buffer</b>	Buffer containing the read entries. The entries are placed directly after each other, and the format of each entry can be seen below. The required size of the buffer depends on the type of connection: Cable (standard data size): 240 bytes. Remote (standard data size): 480 bytes. When large data is enabled (see rmonEnableLargeData()): Cable (large data size): 1024 bytes.																												
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData, rmonNoMoreData Entry format:																												
	<table border="1"> <thead> <tr> <th>Data type</th> <th>Size (byte)</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>signed char</td> <td>1</td> <td>Year relative to year 2000</td> </tr> <tr> <td>unsigned char</td> <td>1</td> <td>Month, 1..12</td> </tr> <tr> <td>unsigned char</td> <td>1</td> <td>Date, 1..31</td> </tr> <tr> <td>unsigned char</td> <td>1</td> <td>Hour, 0..23</td> </tr> <tr> <td>unsigned char</td> <td>1</td> <td>Minute, 0..59</td> </tr> <tr> <td>unsigned char</td> <td>1</td> <td>Second, 0..59</td> </tr> <tr> <td>unsigned char</td> <td>1</td> <td>Tag</td> </tr> <tr> <td>int[n]</td> <td>0-32</td> <td>Log values, where n is the number of values in each entry (0-8)</td> </tr> </tbody> </table>		Data type	Size (byte)	Description	signed char	1	Year relative to year 2000	unsigned char	1	Month, 1..12	unsigned char	1	Date, 1..31	unsigned char	1	Hour, 0..23	unsigned char	1	Minute, 0..59	unsigned char	1	Second, 0..59	unsigned char	1	Tag	int[n]	0-32	Log values, where n is the number of values in each entry (0-8)
Data type	Size (byte)	Description																											
signed char	1	Year relative to year 2000																											
unsigned char	1	Month, 1..12																											
unsigned char	1	Date, 1..31																											
unsigned char	1	Hour, 0..23																											
unsigned char	1	Minute, 0..59																											
unsigned char	1	Second, 0..59																											
unsigned char	1	Tag																											
int[n]	0-32	Log values, where n is the number of values in each entry (0-8)																											

<b>rmonLogGetValuesPerRecord()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State		
<b>Synopsis</b>	rmonRet __stdcall rmonLogGetValuesPerRecord(HRMONCON hCon, int* numberofvalues)			
<b>Description</b>	This function returns information about how many values (up to 8) are stored at each record in the datalogger (is configured via the VPL program in the RTCU device)			
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> </table>		<b>hCon</b>	Handle to connection
<b>hCon</b>	Handle to connection			
<b>Output</b>	<table border="1"> <tr> <td><b>numberofvalues</b></td> <td>The number of values stored in each record in the datalogger.</td> </tr> </table>		<b>numberofvalues</b>	The number of values stored in each record in the datalogger.
<b>numberofvalues</b>	The number of values stored in each record in the datalogger.			
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError			

<b>rmonLogClear()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State		
<b>Synopsis</b>	rmonRet __stdcall rmonLogClear(HRMONCON hCon)			
<b>Description</b>	Clears data in the RTCU datalogger. The current datastructure in the RTCU datalogger is maintained.			
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> </table>		<b>hCon</b>	Handle to connection
<b>hCon</b>	Handle to connection			
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNotInit			

<b>rmonLogGotoLinsec()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State						
<b>Synopsis</b>	rmonRet __stdcall rmonLogGotoLinsec(HRMONCON hCon, struct rmonRTCTime timestamp, unsigned char direction)							
<b>Description</b>	rmonLogGotoLinsec will search for an entry in the datalogger, that matches the specified timestamp, and if no match is found, it will select the nearest record (if any). It is possible to specify the search direction as either forward or backward.							
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>timestamp</b></td> <td>Complete time of record to search for. Please refer to the definition of rmonRTCTime below</td> </tr> <tr> <td><b>direction</b></td> <td>False (0) means backwards search, True (different from 0) means forward search</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>timestamp</b>	Complete time of record to search for. Please refer to the definition of rmonRTCTime below	<b>direction</b>	False (0) means backwards search, True (different from 0) means forward search
<b>hCon</b>	Handle to connection							
<b>timestamp</b>	Complete time of record to search for. Please refer to the definition of rmonRTCTime below							
<b>direction</b>	False (0) means backwards search, True (different from 0) means forward search							
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonNoData							

<b>rmonLogReadByTag()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State	
<b>Synopsis</b>	rmonRet __stdcall rmonLogReadByTag(HRMONCON hCon, int operation, unsigned char tag, int* values_per_rec, int* entries_in_buffer, char* buffer);		
<b>Description</b>	This function reads up to 146 log entries from the datalogger each time it is called. When it is called, the current read pointer is either incremented or decremented, according to the operation parameter.		
<b>Input</b>	<b>hCon</b>	Handle to connection	
	<b>operation</b>	RMONLOGGET_NEXT or RMONLOGGET_PREV	
	<b>tag</b>	The tag that is used to filter datalog entries	
<b>Output</b>	<b>values_per_rec</b>	Number of values in each log entry, maximum 8	
	<b>entries_in_buffer</b>	Number of entries in the buffer, maximum 146	
	<b>buffer</b>	Buffer containing the read entries. The entries are placed directly after each other, and the format of each entry can be seen below. The required size of the buffer depends on the type of connection: Cable (standard data size): 240 bytes. Remote (standard data size): 480 bytes. When large data is enabled (see rmonEnableLargeData()): Cable (large data size): 1024 bytes.	
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData, rmonNoMoreData		
	Entry format:		
	<b>Data type</b>	<b>Size (byte)</b>	<b>Description</b>
	signed char	1	Year relative to year 2000
	unsigned char	1	Month, 1..12
	unsigned char	1	Date, 1..31
	unsigned char	1	Hour, 0..23
	unsigned char	1	Minute, 0..59
	unsigned char	1	Second, 0..59
	unsigned char	1	Tag
	int[n]	0-32	Log values, where n is the number of values in each entry (0-8)

<b>rmonLogSeek()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonLogSeek(HRMONCON hCon, short tag, short n)	
<b>Description</b>	rmonLogSeek will search for an entry in the datalogger, that matches the specified tag, and move n records from there. The n parameter determines the direction of the search.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>tag</b>	The tag to search for.
	<b>n</b>	The number of records to move. > 0 (zero): Seek forward. = 0 (zero): No effect. < 0 (zero): Seek Backwards.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonNoData	

## I/O system functions

This group of functions allows access to the physical in- and outputs of the RTCU device, as well as the memory I/O system. The memory I/O system is accessible through the VPL program as normal VAR\_INPUT/VAR\_OUTPUT variables. The variables must be configured in the RTCU M2M Studio job configuration as either “To memory” or “From Memory”, depending on if they are declared as VAR\_INPUT or VAR\_OUTPUT variables.

The memory I/O system is 4096 elements.

<b>rmonReadIOMemory()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonReadIOMemory(HRMONCON hCon, int location, int count, int type, void *data);	
<b>Description</b>	This function read from the memory I/O system in the RTCU. It is possible to indicate what type of data is stored at each location read; this is to help the function minimizing communication traffic.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>location</b>	This is the start location to read from, 0 based.
	<b>count</b>	Number of memory locations to read
	<b>type</b>	1=BOOL, 2=SINT, 3=INT, 4=DINT, this is the type of data to read from each location
<b>Output</b>	<b>data</b>	Data read from the RTCU will be stored in this buffer (must be ‘count’ number long, and of the same type at ‘type’ specifies (BOOL and SINT is 1 byte, INT is 2 bytes and DINT is 4 bytes) The required size of the buffer depends on the type of connection: Cable (standard data size): 240 bytes. Remote (standard data size): 480 bytes. When large data is enabled (see rmonEnableLargeData()): Cable (large data size): 1024 byte.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonWriteIOMemory()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State										
<b>Synopsis</b>	rmonRet __stdcall rmonWriteIOMemory(HRMONCON hCon, int location, int count, int type, void *data);											
<b>Description</b>	This function writes to the memory I/O system in the RTCU. It is possible to indicate what type of data is to be stored at each location; this is to help the function minimizing communication traffic. Please note that if the location(s) written to, is also used as VAR_OUTPUT variables by the VPL program in the RTCU device, the RTCU and this function will both write to the same location, in which case the writing done by this function, will be overwritten by the RTCU device itself (all I/O configured variables will be updated in each scan of the VPL program in the RTCU device).											
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>location</b></td> <td>This is the start location to write to, 0 based.</td> </tr> <tr> <td><b>count</b></td> <td>Number of memory locations to write</td> </tr> <tr> <td><b>type</b></td> <td>1=BOOL, 2=SINT, 3=INT, 4=DINT</td> </tr> <tr> <td><b>data</b></td> <td>Data written to the RTCU is taken from this buffer (must be 'count' number long, and of the same type as 'type' specifies (BOOL and SINT is 1 byte, INT is 2 bytes and DINT is 4 bytes) The required size of the buffer depends on the type of connection: Cable (standard data size): 240 bytes. Remote (standard data size): 480 bytes. When large data is enabled (see rmonEnableLargeData()): Cable (large data size): 1024 byte.</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>location</b>	This is the start location to write to, 0 based.	<b>count</b>	Number of memory locations to write	<b>type</b>	1=BOOL, 2=SINT, 3=INT, 4=DINT	<b>data</b>	Data written to the RTCU is taken from this buffer (must be 'count' number long, and of the same type as 'type' specifies (BOOL and SINT is 1 byte, INT is 2 bytes and DINT is 4 bytes) The required size of the buffer depends on the type of connection: Cable (standard data size): 240 bytes. Remote (standard data size): 480 bytes. When large data is enabled (see rmonEnableLargeData()): Cable (large data size): 1024 byte.
<b>hCon</b>	Handle to connection											
<b>location</b>	This is the start location to write to, 0 based.											
<b>count</b>	Number of memory locations to write											
<b>type</b>	1=BOOL, 2=SINT, 3=INT, 4=DINT											
<b>data</b>	Data written to the RTCU is taken from this buffer (must be 'count' number long, and of the same type as 'type' specifies (BOOL and SINT is 1 byte, INT is 2 bytes and DINT is 4 bytes) The required size of the buffer depends on the type of connection: Cable (standard data size): 240 bytes. Remote (standard data size): 480 bytes. When large data is enabled (see rmonEnableLargeData()): Cable (large data size): 1024 byte.											
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonError											

<b>rmonGetIOState()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State																					
<b>Synopsis</b>	rmonRet __stdcall rmonGetIOState(HRMONCON hCon, int iotype, int ioindex, int* value)																						
<b>Description</b>	This function is used to read the state of the physical in- and output signals in the RTCU device. The 'iotype' indicates which input/output you are reading from, and 'ioindex' indicates which input- or output number you are reading.																						
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>iotype</b></td> <td>Select the type of I/O system you want to read from, see below</td> </tr> <tr> <td><b>ioindex</b></td> <td>Valid index of IO to get value from. Starts with index 0</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>iotype</b>	Select the type of I/O system you want to read from, see below	<b>ioindex</b>	Valid index of IO to get value from. Starts with index 0															
<b>hCon</b>	Handle to connection																						
<b>iotype</b>	Select the type of I/O system you want to read from, see below																						
<b>ioindex</b>	Valid index of IO to get value from. Starts with index 0																						
<b>Output</b>	<table border="1"> <tr> <td><b>value</b></td> <td>Input or output value</td> </tr> </table>		<b>value</b>	Input or output value																			
<b>value</b>	Input or output value																						
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError																						
	iotype:																						
	<table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>RMON_IOTYPE_DIN</td> <td>1</td> <td>Digital input</td> </tr> <tr> <td>RMON_IOTYPE_DOUT</td> <td>2</td> <td>Digital output</td> </tr> <tr> <td>RMON_IOTYPE_AIN</td> <td>3</td> <td>Analog input</td> </tr> <tr> <td>RMON_IOTYPE_AOUT</td> <td>4</td> <td>Analog output</td> </tr> <tr> <td>RMON_IOTYPE_LED</td> <td>5</td> <td>LED</td> </tr> <tr> <td>RMON_IOTYPE_DIPSW</td> <td>6</td> <td>Dip switch</td> </tr> </tbody> </table>		Symbolic name	Value	Description	RMON_IOTYPE_DIN	1	Digital input	RMON_IOTYPE_DOUT	2	Digital output	RMON_IOTYPE_AIN	3	Analog input	RMON_IOTYPE_AOUT	4	Analog output	RMON_IOTYPE_LED	5	LED	RMON_IOTYPE_DIPSW	6	Dip switch
Symbolic name	Value	Description																					
RMON_IOTYPE_DIN	1	Digital input																					
RMON_IOTYPE_DOUT	2	Digital output																					
RMON_IOTYPE_AIN	3	Analog input																					
RMON_IOTYPE_AOUT	4	Analog output																					
RMON_IOTYPE_LED	5	LED																					
RMON_IOTYPE_DIPSW	6	Dip switch																					

<b>rmonSetIOState()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State													
<b>Synopsis</b>	rmonRet __stdcall rmonSetIOState(HRMONCON hCon, int iotype, int ioindex, int value)														
<b>Description</b>	<p>This function is used to set the status of the physical output signals in the RTCU device. The 'iotype' indicated which Output you are writing to, and 'ioindex' indicates which output number you are writing to.</p> <p>Please note that if the output written to, is also used as a VAR_OUTPUT variable by the VPL program in the RTCU device, the RTCU and this function will both write to the same output, in which case the writing done by this function will be overwritten by the RTCU device itself (all I/O configured variables will be updated in each scan of the VPL program in the RTCU device).</p>														
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td colspan="2"><b>Handle to connection</b></td> </tr> <tr> <td><b>iotype</b></td> <td colspan="2">Select the type of output system you want to set, see below</td> </tr> <tr> <td><b>ioindex</b></td> <td colspan="2">This is the number of the output, 0 based.</td> </tr> <tr> <td><b>value</b></td> <td colspan="2">Value to set</td> </tr> </table>			<b>hCon</b>	<b>Handle to connection</b>		<b>iotype</b>	Select the type of output system you want to set, see below		<b>ioindex</b>	This is the number of the output, 0 based.		<b>value</b>	Value to set	
<b>hCon</b>	<b>Handle to connection</b>														
<b>iotype</b>	Select the type of output system you want to set, see below														
<b>ioindex</b>	This is the number of the output, 0 based.														
<b>value</b>	Value to set														
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError														
	iotype:														
	<table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>RMON_IOTYPE_DOUT</td> <td>2</td> <td>Digital output</td> </tr> <tr> <td>RMON_IOTYPE_AOUT</td> <td>4</td> <td>Analog output</td> </tr> <tr> <td>RMON_IOTYPE_LED</td> <td>5</td> <td>LED</td> </tr> </tbody> </table>			Symbolic name	Value	Description	RMON_IOTYPE_DOUT	2	Digital output	RMON_IOTYPE_AOUT	4	Analog output	RMON_IOTYPE_LED	5	LED
Symbolic name	Value	Description													
RMON_IOTYPE_DOUT	2	Digital output													
RMON_IOTYPE_AOUT	4	Analog output													
RMON_IOTYPE_LED	5	LED													

<b>rmonGetIOCount()</b>		<b>RTCU architecture:</b> X32, NX32 <b>Called in:</b> Connected State				
<b>Synopsis</b>	rmonRet __stdcall rmonGetIOCount(HRMONCON hCon, struct rmonIOCount *data)					
<b>Description</b>	This function is used to read the number of inputs and outputs on the device. (Onboard and external)					
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td colspan="2">Handle to connection</td> </tr> </table>			<b>hCon</b>	Handle to connection	
<b>hCon</b>	Handle to connection					
<b>Output</b>	<table border="1"> <tr> <td><b>data</b></td> <td colspan="2">A structure that contains the I/O count</td> </tr> </table>			<b>data</b>	A structure that contains the I/O count	
<b>data</b>	A structure that contains the I/O count					
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError					
	<pre>typedef struct {     unsigned char    NumberOfAI;     unsigned char    NumberOfAO;     unsigned char    NumberOfDI;     unsigned char    NumberOfDO;     unsigned char    NumberOfDIPSW;     unsigned char    NumberOfLED;     unsigned char    NumberOfExtAI;     unsigned char    NumberOfExtAO;     unsigned char    NumberOfExtDI;     unsigned char    NumberOfExtDO;     unsigned char    NumberOfExtDIPSW;     unsigned char    NumberOfExtLED; } rmonIOCount;</pre>					

## Real time clock

Functions that will read and set the realtime clock in the RTCU device.

The two functions, rmonGetRTC() and rmonSetRTC, both uses the following structure:

```
struct rmonRTCTime {
    unsigned short year;        // 2000..2048
    unsigned char  month;      // 01..12
    unsigned char  date;       // 01..31
    unsigned char  day;        // 01..07
    unsigned char  hour;       // 00..23
    unsigned char  minute;     // 00..59
    unsigned char  second;     // 00..59
};
```

<b>rmonGetRTC()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonGetRTC(HRMONCON hCon, struct rmonRTCTime *rtc)	
<b>Description</b>	Reads the real time clock on the RTCU device. Returns the current time in a rmonRTCTime structure.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>rtc</b>	Please refer to the definition of rmonRTCTime above
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonSetRTC()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonSetRTC(HRMONCON hCon, struct rmonRTCTime *rtc)	
<b>Description</b>	Sets the real time clock on the RTCU device. The time is supplied in a rmonRTCTime structure.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>rtc</b>	Please refer to the definition of rmonRTCTime above
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

## GSM/SMS functions

This is a set of functions, which allows you to read and set various parameters used in the RTCU devices interaction with the GSM module.

The functions also allow you to send and receive “fake” SMS messages to/from the RTCU device. If the VPL program in the device sends an SMS message to phone number “9999”, using the VPL function `gsmSendSMS()` / `gsmSendPDU()`, the message will be received by the library, and the message will then be available through the function `rmonReceiveSMS()`. The same goes for your application, it can call `rmonSendSMS()`, and the VPL program in the RTCU can use `gsmIncomingSMS()` / `gsmIncomingPDU()` to receive this message sent from your application. This is a very easy to use way of communicating small messages back and forth between your PC application and the VPL application of the RTCU device.

<b>rmonGetIMEI()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	<code>rmonRet __stdcall rmonGetIMEI(HRMONCON hCon, char *IMEInumber, int bufsize)</code>	
<b>Description</b>	rmonGetIMEI is used for fetching the IMEI number of the GSM module	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>bufsize</b>	Number of characters to read. If bufsize exceeds the number of characters in the number, only the number of characters present will be put in the output buffer.
<b>Output</b>	<b>IMEInumber</b>	This is where the information will be stored
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonGetIMSI()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	<code>rmonRet __stdcall rmonGetIMSI(HRMONCON hCon, char *IMSInumber, int bufsize)</code>	
<b>Description</b>	rmonGetIMSI is used for fetching the IMSI number of the GSM module	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>bufsize</b>	Number of characters to read. If bufsize exceeds the number of characters in the number, only the number of characters present will be put in the output buffer.
<b>Output</b>	<b>IMSInumber</b>	This is where the information will be stored
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonGetICCID()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	<code>rmonRet __stdcall rmonGetICCID(HRMONCON hCon, char *ICCIDnumber, int bufsize)</code>	
<b>Description</b>	rmonGetICCID is used for fetching the ICCID number of the GSM module	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>bufsize</b>	Number of characters to read. If bufsize exceeds the number of characters in the number, only the number of characters present will be put in the output buffer.
<b>Output</b>	<b>ICCIDnumber</b>	This is where the information will be stored

**Returns** rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError

<b>rmonSendSMS()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State	
<b>Synopsis</b>	rmonRet __stdcall rmonSendSMS(HRMONCON hCon, int smstype, int messageLength, const char* message)		
<b>Description</b>	The PC application can send “fake” SMS messages to the RTCU device using this function. The RTCU will receive the SMS messages with gsmIncomingSMS() / gsmIncomingPDU(), and when a message is sent to the RTCU using this function, the phonenummer variable in gsmIncomingSMS() / gsmIncomingPDU() will indicate “9999” as the originator of the message.		
<b>Input</b>	<b>hCon</b>	Handle to connection	
	<b>smstype</b>	Type of SMS message received, see below	
	<b>messageLength</b>	Only used when smstype is RMONSMS_BINARY	
	<b>message</b>	Zero terminated AZCII string when smstype is RMONSMS_TEXT. If smstype is RMONSMS_BINARY messageLength specifies length of data in message.	
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError		
	smstype:		
	<b>Symbolic name</b>	<b>Value</b>	<b>Description</b>
	RMONSMS_TEXT	0	Text based SMS message
	RMONSMS_BINARY	1	Binary SMS message

<b>rmonReceiveSMS()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State	
<b>Synopsis</b>	rmonRet __stdcall rmonReceiveSMS(HRMONCON hCon, int* smstype, int* dataLength, char* data)		
<b>Description</b>	When the connected RTCU device sends an SMS message to phone number “9999” using either gsmSendSMS() or gsmSendPDU(), this function will receive these messages. Please note that this function is blocking, it will not return until a message is received.		
<b>Input</b>	<b>gCon</b>	Handle to connection	
	<b>Output</b>	<b>smstype</b>	Type of SMS message received, see below
	<b>dataLength</b>	Only used when smstype = RMONSMS_BINARY	
	<b>data</b>	Zero terminated ASCII string when smstype = RMONSMS_TEXT	
<b>Returns</b>	rmonOK, rmonIllegalHandle, rmonTargetError, rmonNoData		
	smstype:		
	<b>Symbolic name</b>	<b>Value</b>	<b>Description</b>
	RMONSMS_TEXT	0	Text based SMS message
	RMONSMS_BINARY	1	Binary SMS message

<b>rmonReceiveSMSEnable()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State				
<b>Synopsis</b>	rmonRet __stdcall rmonReceiveSMSEnable(HRMONCON hCon, int enable)					
<b>Description</b>	Using this function, it is possible to either enable or disable reception of the “fake” SMS messages from the RTCU device.  Note, if disabling, rmonReceiveSMS() will still block, waiting for a message to arrive.					
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>enable</b></td> <td>0=disable, 1=enable</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>enable</b>	0=disable, 1=enable
<b>hCon</b>	Handle to connection					
<b>enable</b>	0=disable, 1=enable					
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError					

<b>rmonGetGSMSignalLevel()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State		
<b>Synopsis</b>	rmonRet __stdcall rmonGetGSMSignalLevel(HRMONCON hCon, int* signal)			
<b>Description</b>	This function returns the GSM signal level. (This function does the same as the VPL function gsmSignalLevel()).			
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> </table>		<b>hCon</b>	Handle to connection
<b>hCon</b>	Handle to connection			
<b>Output</b>	<table border="1"> <tr> <td><b>signal</b></td> <td>The GSM signal strength or 0 (zero) if not connected.</td> </tr> </table>		<b>signal</b>	The GSM signal strength or 0 (zero) if not connected.
<b>signal</b>	The GSM signal strength or 0 (zero) if not connected.			
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError			

<b>rmonSetAllowedCallerList()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State				
<b>Synopsis</b>	rmonRet __stdcall rmonSetAllowedCallerList (HRMONCON hCon, const char numbers[81])					
<b>Description</b>	Sets list of allowed phone numbers that can make incoming data calls to the RTCU. Phone numbers must be separated with the “,” character. (This function does the same as the VPL function gsmSetListOfCallers()).					
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>numbers</b></td> <td>List of phonenumber separated by “,” character</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>numbers</b>	List of phonenumber separated by “,” character
<b>hCon</b>	Handle to connection					
<b>numbers</b>	List of phonenumber separated by “,” character					
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError					

<b>rmonGetAllowedCallerList()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonGetAllowedCallerList(HRMONCON hCon, char numbers[81])	
<b>Description</b>	Fetches list of allowed caller numbers set in rmonSetAllowedCallerList(). (This function does the same as the VPL function gsmGetListOfCallers()).	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>numbers</b>	List of allowed caller numbers separated by “,” character.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonSetGSMPIN()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonSetGSMPIN (HRMONCON hCon, const char pin[5])	
<b>Description</b>	Sets the SIM PIN code to use for the SIM card in the RTCU device. This does NOT change the PIN code on the SIM card; it simply tells the RTCU device which PIN code to use when powering up the GSM module !  An empty string will disable use of PIN code (SIM PIN code must be disabled on the SIM card, use a normal mobile telephone for doing this) Specifying a wrong GSM pin code will cause a RTCU fault.  <b>If the RTCU is restarted more than 3 times with the wrong SIM pin code, the SIM card will be locked, and it must be unlocked in a normal mobile phone, using the GSM operator supplied PUK code !</b>  Please notice the SIM PIN code may also be set using the RTCU M2M Studio (menu: Device -> Configuration -> Cellular options) (This function does the same as the VPL function gsmSetPin()).	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>pin</b>	New SIM PIN code to be set
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonGetGSMPIN()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonGetGSMPIN (HRMONCON hCon, const char pin[5])	
<b>Description</b>	Fetches the GSM SIM PIN code from the RTCU (see rmonSetGSMPin() above). An empty string denotes that the PIN code has been disabled.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>pin</b>	Current SIM PIN code. Empty string specifies the PIN code to be currently disabled
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

## Filesystem functions

This is a set of functions that offers a broad range of operations on the file system present in the RTCU device.

The filesystem error-codes start from 100, but are otherwise identical to the VPL ones:

Symbolic name	Value	Description
RMONFS_INVALIDDRIVE	101	The media is not opened
RMONFS_NOTFOUND	105	The directory or file is not found
RMONFS_DUPLICATED	106	The directory or file already exist
RMONFS_NOMOREENTRY	107	The media is full
RMONFS_NOTOPEN	108	The file is not open
RMONFS_LOCKED	112	The file is in use
RMONFS_NOTEMPTY	114	The directory is not empty
RMONFS_CARDREMOVED	116	The media is not present
RMONFS_ONDRIVE	117	Media communication error
RMONFS_BUSY	122	The media is busy
RMONFS_WRITEPROTECT	123	The media is write-protected
RMONFS_FILEACCESS	138	The file is no longer accessible and must be closed
RMONFS_EXTENSION	139	The media is busy with Platform extension

The filesystem functions have these limitations in addition to the ones for the file system in general (See the RTCU M2M Studio Online-help/Manual):

A client can only have one open file at any given time. If more files are opened, the already open file is closed.

The working directory is shared with all other clients connected to the device. Because of this it is recommended to use absolute paths where possible.

The media available:

Media ID	Drive	Description
0	A:	The SD-CARD.
1	B:	The Internal drive.
2	P:	The Intellisync Project drive.
3	U:	The USB Mass storage drive. *

\* The USB Mass storage drive is only available on NX32L devices with an USB host port.

<b>rmonMediaPresent()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonMediaPresent(HRMONCON hCon, int media, int* state, int* fserr)	
<b>Description</b>	Queries whether the Media is present or not.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>media</b>	The media ID. (See introduction for available media)
<b>Output</b>	<b>State</b>	=0 (zero) if media is not present. <>0 (zero) if media is present
	<b>Fserr</b>	Error code from the filesystem
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonMediaWriteprotected()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonMediaWriteprotected (HRMONCON hCon, int media, int* state, int* fserr)	
<b>Description</b>	Queries whether the Media is write protected or not.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>media</b>	The media ID. (See introduction for available media)
<b>Output</b>	<b>State</b>	=0 (zero) if media is not write protected. <>0 (zero) if media is write protected.
	<b>Fserr</b>	Error code from the filesystem
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonMediaOpen()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonMediaopen (HRMONCON hCon, int media, int* fserr)	
<b>Description</b>	Open the media for use with the filesystem.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>media</b>	The media ID. (See introduction for available media)
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonMediaClose()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonMediaopen (HRMONCON hCon, int media, int* fserr)	
<b>Description</b>	Close the media for use with the filesystem.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>media</b>	The media ID. (See introduction for available media)
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonMediaQuickformat()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonMediaQuickformat (HRMONCON hCon, int media, int* fserr)	
<b>Description</b>	Quick formats the media.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>media</b>	The media ID. (See introduction for available media)
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonMediaQuickformatX()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonMediaQuickformatX (HRMONCON hCon, int media, int flags, int* fserr)	
<b>Description</b>	Quick formats the media.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>media</b>	The media ID. (See introduction for available media)
	<b>flags</b>	Option flags to control the behaviour of the format. (Only used by NX32L devices)
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	
	flags:	
	<b>Symbolic name</b>	<b>Value</b> <b>Description</b>
	<b>RMONFS_FMTFLAG_FULL</b>	1          Perform full format of drive

<b>rmonMediaEject()</b>		<b>RTC architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonMediaEject(HRMONCON hCon, int media, int* fserr)	
<b>Description</b>	Eject the media.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>media</b>	The media ID. (See introduction for available media)
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonMediaInformation()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State										
<b>Synopsis</b>	rmonRet __stdcall rmonMediaInformation(HRMONCON hCon, struct rmonMediaInfo media[8])											
<b>Description</b>	<p>Queries the device for which media is mounted, what type of media it is, and the capacity of the media.</p> <p>The media type can be one of the following:</p> <table border="1"> <tr><td>0</td><td>Not mounted</td></tr> <tr><td>1</td><td>SD-CARD</td></tr> <tr><td>2</td><td>Internal FLASH</td></tr> <tr><td>3</td><td>Intellisync Project Drive</td></tr> <tr><td>4</td><td>USB Mass storage drive</td></tr> </table>		0	Not mounted	1	SD-CARD	2	Internal FLASH	3	Intellisync Project Drive	4	USB Mass storage drive
0	Not mounted											
1	SD-CARD											
2	Internal FLASH											
3	Intellisync Project Drive											
4	USB Mass storage drive											
<b>Input</b>	<b>hCon</b>	Handle to connection										
<b>Output</b>	<b>media</b>	An array of media information structures.										
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError											
	<pre> struct rmonMediaInfo {     unsigned char    Type;    // Type of media     unsigned long    Size;    // Size of the media, lower 32bits     unsigned long    SizeHi; // Size of the media, upper 32bits }; </pre>											

<b>rmonMediaSize()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonMediaSize(HRMONCON hCon, int media, unsigned long* SizeLo, unsigned long* SizeHi, unsigned long* FreeLo, unsigned long* FreeHi)	
<b>Description</b>	Retrieve the total size and free size of a media.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>media</b>	The media ID. (See introduction for available media)
<b>Output</b>	<b>SizeLo</b>	The total size of the media, lower 32 bits.
	<b>SizeHi</b>	The total size of the media, upper 32 bits.
	<b>FreeLo</b>	The free size of the media, lower 32 bits.
	<b>FreeHi</b>	The free size of the media, upper 32 bits.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFSStatusLED()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFSStatusLED(HRMONCON hCon, int enable, int* fserr)	
<b>Description</b>	Using this function, it is possible to either enable or disable the filesystem status LED's.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Enable</b>	0=disable, 1=enable
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonDirCreate()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonDirCreate(HRMONCON hCon, const char name[61], int* fserr)	
<b>Description</b>	Create a new directory.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Name</b>	Name of the directory to create. (60 characters + 0 (zero) terminator) Both absolute and relative path can be used.
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonDirChange()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonDirChange(HRMONCON hCon, const char path[61], int* fserr)	
<b>Description</b>	Change the working directory.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Path</b>	Path to the new working directory. (60 characters + 0 (zero) terminator) Both absolute and relative path can be used.
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonDirCurrent()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonDirCurrent(HRMONCON hCon, char path[61], int* fserr)	
<b>Description</b>	Retrieve the absolute path to the working directory	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>Path</b>	The absolute path to the working directory. (60 characters + 0 (zero) terminator)
	<b>Fserr</b>	Error code from the file system.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonDirCatalog()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonDirCatalog(HRMONCON hCon, short index, char name[15], struct rmonRTCTime* time, long* length, int* fserr)	
<b>Description</b>	Retrieves the information of an entry in the working directory.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Index</b>	The index of the directory entry to retrieve.
<b>Output</b>	<b>Name</b>	The name of the directory entry. (14 characters + zero terminator)
	<b>Time</b>	The creation time of the file. Uses the same structure as rmonGetRTC. Not used for directories.
	<b>Length</b>	The size of the file. Not used for directories.
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonDirCatalogX()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State									
<b>Synopsis</b>	rmonRet __stdcall rmonDirCatalog(HRMONCON hCon, char* wild, rmoncbdirentry pfunc, void* uptr, int* fserr)										
<b>Description</b>	Retrieves the information of all the entries in the working directory that matches the provided wildcard string										
<b>Input</b>	<b>hCon</b>	Handle to connection									
	<b>wild</b>	Wildcard string to search for. The wildcards ? for any one character and * for any number of characters are supported.									
	<b>pfunc</b>	Pointer to callback function which is called for each directory item.									
	<b>uptr</b>	Pointer to user argument used in callback function									
<b>Output</b>	<b>fserr</b>	Error code from the filesystem.									
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonError, rmonCancelled										
	<p>The call-back function is defined as follows:</p> <pre>typedef int (__stdcall *rmoncbdirentry)(void* uptr, const char* filename, unsigned char type, long timestamp, unsigned long size);</pre> <p>Type:</p> <table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>RMONFS_TYPE_FILE</td> <td>0</td> <td>File</td> </tr> <tr> <td>RMONFS_TYPE_FOLDER</td> <td>1</td> <td>Directory</td> </tr> </tbody> </table>		Symbolic name	Value	Description	RMONFS_TYPE_FILE	0	File	RMONFS_TYPE_FOLDER	1	Directory
Symbolic name	Value	Description									
RMONFS_TYPE_FILE	0	File									
RMONFS_TYPE_FOLDER	1	Directory									

<b>rmonDirDelete()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonDirDelete(HRMONCON hCon, const char name[61], int* fserr)	
<b>Description</b>	Delete a directory.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Name</b>	The name of the directory. (60 characters + zero terminator) Both absolute and relative path can be used.
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileCreate()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileCreate(HRMONCON hCon, const char name[61], int* fserr)	
<b>Description</b>	Creates a new file. If a file is already open, it will be closed before the new file is created.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Name</b>	The name of the file to create. (60 characters + zero terminator) Both absolute and relative path can be used.
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileOpen()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileOpen(HRMONCON hCon, const char name[61], int* fserr)	
<b>Description</b>	Opens a file. If a file is already open, it will be closed before the new file is opened.	
<b>Input</b>	<b>HCon</b>	Handle to connection
	<b>Name</b>	The name of the file to create. (60 characters + zero terminator) Both absolute and relative path can be used.
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileExists()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileExists(HRMONCON hCon, const char name[61], char* state, int* fserr)	
<b>Description</b>	Query whether a file exists.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Name</b>	The name of the file to create. (60 characters + zero terminator) Both absolute and relative path can be used.
<b>Output</b>	<b>State</b>	=0 (zero) if file does not exist. <>0 (zero) if file does exist.
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileRename()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileRename(HRMONCON hCon, const char name_old[61], const char name_new[13], int* fserr)	
<b>Description</b>	Renames a file	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Name_new</b>	The new name of the file. (12 characters + zero terminator)
	<b>Name_old</b>	The name of the file to rename. Both absolute and relative paths can be used (60 characters + zero terminator)
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileDelete()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileDelete(HRMONCON hCon, const char name[61], int* fserr)	
<b>Description</b>	Delete a file.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Name</b>	The name of the file to create. (60 characters + zero terminator) Both absolute and relative path can be used.
<b>Output</b>	<b>State</b>	=0 (zero) if file does not exist. <>0 (zero) if file does exist.
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileStatus()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileStatus(HRMONCON hCon, int* status, int* fserr)	
<b>Description</b>	Retrieve the status of the open file.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>Status</b>	The status of the file. Identical to the return value of the fsFileStatus VPL function.
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileGetInfo()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileGetInfo(HRMONCON hCon, const char name[61], struct rmonRTCTime* time, long* length, int* fserr)	
<b>Description</b>	Retrieve the size and creation timestamp of a file.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>name</b>	The name of the file. (60 characters + zero terminator) Both absolute and relative path can be used.
<b>Output</b>	<b>Time</b>	The creation timestamp. Please refer to the definition of rmonRTCTime above (Page 40)
	<b>Length</b>	The size of the file in bytes.
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileSeek()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileSeek(HRMONCON hCon, long offset, int* fserr)	
<b>Description</b>	Moves the file pointer.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Offset</b>	The new position relative to the Start of file. >0 (zero) – Position in file. =0 (zero) – Start of file. -1 – End of file.
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFilePosition()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFilePosition(HRMONCON hCon, long* position, int* fserr)	
<b>Description</b>	Retrieve the file pointer position of the open file.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>Position</b>	The file pointer position
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileRead()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileRead(HRMONCON hCon, int elemcnt, char* buffer, int* elemread, int* fserr, rmoncbprogress pfunc, void* uptr)	
<b>Description</b>	Read a block of data from file.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>Elemcnt</b>	The number of bytes to read from file.
	<b>Pfunc</b>	Pointer to function where progress is reported.
	<b>Arg</b>	Pointer to user argument used when reporting progress.
<b>Output</b>	<b>Buffer</b>	The buffer where the data read from the file is stored.
	<b>elemread</b>	The number of bytes read from file.
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileReadString()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileReadString(HRMONCON hCon, char str[241], int* elemread, int* fserr)	
<b>Description</b>	Reads a string from the file. The function will read until a <CR><LF> termination sequence is found or the buffer is full (240 characters).	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>str</b>	The buffer where the string read from the file is stored. (240 characters + zero terminator)
	<b>elemread</b>	The number of bytes read from file.
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileWrite()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileWrite(HRMONCON hCon, int elemcnt, char* buffer, int* elemwr, int* fserr, rmoncbprogress pfunc, void* uptr)	
<b>Description</b>	Write a block of data to file.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>elemcnt</b>	The number of bytes to write to file.
	<b>Buffer</b>	The buffer where the data to write is stored.
	<b>pfunc</b>	Pointer to function where progress is reported.
	<b>uptr</b>	Pointer to user argument used when reporting progress.
<b>Output</b>	<b>elemwr</b>	The number of bytes written to file
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileWriteString()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileWriteString(HRMONCON hCon, const char str[241], int* elemwr, int* fserr)	
<b>Description</b>	Write a string to file.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>str</b>	The string to write to file. (240 characters + zero terminator)
<b>Output</b>	<b>elemwr</b>	The number of bytes written to file
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileWriteStringNL()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileWriteStringNL(HRMONCON hCon, const char str[241], int* elemwr, int* fserr)	
<b>Description</b>	Write a string to file. <CR><LF> are appended.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>str</b>	The string to write to file. (240 characters + zero terminator)
<b>Output</b>	<b>elemwr</b>	The number of bytes written to file
	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileClose()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileClose(HRMONCON hCon, int* fserr)	
<b>Description</b>	Close the file.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFileFlush()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFileFlush(HRMONCON hCon, int* fserr)	
<b>Description</b>	Flush cached write operations to media.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>Fserr</b>	Error code from the filesystem.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

## Security functions

The certificates of the RTCU device can be manipulated by this set of functions.

For a more detailed description of security in the RTCU devices, please refer to the online help for the RTCU M2M Studio.

<b>rmonSecurityImport()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State										
<b>Synopsis</b>	rmonRet __stdcall rmonSecurityImport(HRMONCON hCon, const char* name, const char* filename_cert, const char* filename_key, const int replace)											
<b>Description</b>	Import a certificate to the device.											
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>name</b></td> <td>The name of the certificate</td> </tr> <tr> <td><b>filename_cert</b></td> <td>The file name of the certificate to import.</td> </tr> <tr> <td><b>filename_key</b></td> <td>Optional file name of private encryption key to include with the certificate.</td> </tr> <tr> <td><b>replace</b></td> <td>0 = Fail if a certificate with the name already exists, 1 = Replace existing certificate.</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>name</b>	The name of the certificate	<b>filename_cert</b>	The file name of the certificate to import.	<b>filename_key</b>	Optional file name of private encryption key to include with the certificate.	<b>replace</b>	0 = Fail if a certificate with the name already exists, 1 = Replace existing certificate.
<b>hCon</b>	Handle to connection											
<b>name</b>	The name of the certificate											
<b>filename_cert</b>	The file name of the certificate to import.											
<b>filename_key</b>	Optional file name of private encryption key to include with the certificate.											
<b>replace</b>	0 = Fail if a certificate with the name already exists, 1 = Replace existing certificate.											
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonFileNotFound, rmonIllegalFile, rmonError											

<b>rmonSecurityRemove()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State				
<b>Synopsis</b>	rmonRet __stdcall rmonSecurityRemove(HRMONCON hCon, const char* name)					
<b>Description</b>	Remove a certificate from the device.					
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>name</b></td> <td>The name of the certificate</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>name</b>	The name of the certificate
<b>hCon</b>	Handle to connection					
<b>name</b>	The name of the certificate					
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonError					

<b>rmonSecurityInfo()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State								
<b>Synopsis</b>	rmonRet __stdcall rmonSecurityInfo(HRMONCON hCon, const char* name, rmoncbcertificate pfunc, void* arg)									
<b>Description</b>	Fetches information about a certificate from the device.									
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>name</b></td> <td>The name of the certificate</td> </tr> <tr> <td><b>pfunc</b></td> <td>Pointer to callback function which is called with the certificate information</td> </tr> <tr> <td><b>arg</b></td> <td>Pointer to user argument used in callback function</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>name</b>	The name of the certificate	<b>pfunc</b>	Pointer to callback function which is called with the certificate information	<b>arg</b>	Pointer to user argument used in callback function
<b>hCon</b>	Handle to connection									
<b>name</b>	The name of the certificate									
<b>pfunc</b>	Pointer to callback function which is called with the certificate information									
<b>arg</b>	Pointer to user argument used in callback function									
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonError									
	The call-back function is defined as follows:									
	<pre>typedef int (__stdcall *rmoncbcertificate)(void* uptr, const char* name, unsigned char type, unsigned key, const char* subject, const char* issuer, long linsec_from, long linsec_to);</pre>									

## System Object Storage functions

The System Object Storage (SOS) of the RTCU device can be manipulated by this set of functions. For a more detailed description of the SOS in the RTCU devices, please refer to the online help for the RTCU M2M Studio.

Object table:

Symbolic name	Value	Description
RMONOBJ_TABLE_SYSTEM	1	The system object table
RMONOBJ_TABLE_USER	2	The user object table

Object types:

Symbolic name	Value	Description
RMONOBJ_TYPE_BOOL	1	Boolean value
RMONOBJ_TYPE_INT	2	Integer value
RMONOBJ_TYPE_STRING	3	String value
RMONOBJ_TYPE_DATA	4	Binary data value
RMONOBJ_TYPE_FLOAT	5	Floating point value
RMONOBJ_TYPE_DOUBLE	6	Double floating point value

<b>rmonObjectRead()</b>		RTCU architecture: NX32L Called in: Connected State																																	
<b>Synopsis</b>	rmonRet __stdcall rmonObjectRead(HRMONCON hCon, const int table, const char* name, rmoncobject output, void* uptr)																																		
<b>Description</b>	Fetches objects from the device. Can not be used with DOUBLE values, use rmonObjectReadX instead.																																		
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>table</b></td> <td>The object table to use</td> </tr> <tr> <td><b>name</b></td> <td>The name of the object. Wildcards can be used.</td> </tr> <tr> <td><b>output</b></td> <td>Pointer to callback function which is called for each object read</td> </tr> <tr> <td><b>uptr</b></td> <td>Pointer to user argument used in callback function</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>table</b>	The object table to use	<b>name</b>	The name of the object. Wildcards can be used.	<b>output</b>	Pointer to callback function which is called for each object read	<b>uptr</b>	Pointer to user argument used in callback function																							
<b>hCon</b>	Handle to connection																																		
<b>table</b>	The object table to use																																		
<b>name</b>	The name of the object. Wildcards can be used.																																		
<b>output</b>	Pointer to callback function which is called for each object read																																		
<b>uptr</b>	Pointer to user argument used in callback function																																		
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonError, rmonNoData, rmonCancelled																																		
	<p>The call-back function is defined as follows:</p> <pre>typedef int (__stdcall *rmoncobject)(void* uptr, const char* name, int type, int size, unsigned char* data, unsigned long flags, int min_val, int max_val, const char* description);</pre> <p>The callback must return 1 to continue to the next object. Return 0 to stop fetching the objects.</p> <table border="1"> <tr> <td><b>uptr</b></td> <td colspan="2">User pointer</td> </tr> <tr> <td><b>name</b></td> <td colspan="2">The name of the object</td> </tr> <tr> <td><b>type</b></td> <td colspan="2">The type of the object. (See object types above)</td> </tr> <tr> <td><b>size</b></td> <td colspan="2">The size of the data in bytes</td> </tr> <tr> <td><b>data</b></td> <td colspan="2">The object data.</td> </tr> <tr> <td><b>flags</b></td> <td colspan="2">Object flags:</td> </tr> <tr> <td></td> <td><b>0x0001</b></td> <td>Read only object</td> </tr> <tr> <td></td> <td><b>0x0002</b></td> <td>Encrypted object</td> </tr> <tr> <td><b>min_val</b></td> <td colspan="2">The minimum value or the minimum length, of the object.</td> </tr> <tr> <td><b>max_val</b></td> <td colspan="2">The maximum value or the maximum length, of the object.</td> </tr> <tr> <td><b>description</b></td> <td colspan="2">A short description of the object.</td> </tr> </table>		<b>uptr</b>	User pointer		<b>name</b>	The name of the object		<b>type</b>	The type of the object. (See object types above)		<b>size</b>	The size of the data in bytes		<b>data</b>	The object data.		<b>flags</b>	Object flags:			<b>0x0001</b>	Read only object		<b>0x0002</b>	Encrypted object	<b>min_val</b>	The minimum value or the minimum length, of the object.		<b>max_val</b>	The maximum value or the maximum length, of the object.		<b>description</b>	A short description of the object.	
<b>uptr</b>	User pointer																																		
<b>name</b>	The name of the object																																		
<b>type</b>	The type of the object. (See object types above)																																		
<b>size</b>	The size of the data in bytes																																		
<b>data</b>	The object data.																																		
<b>flags</b>	Object flags:																																		
	<b>0x0001</b>	Read only object																																	
	<b>0x0002</b>	Encrypted object																																	
<b>min_val</b>	The minimum value or the minimum length, of the object.																																		
<b>max_val</b>	The maximum value or the maximum length, of the object.																																		
<b>description</b>	A short description of the object.																																		

<b>rmonObjectReadX()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State																																							
<b>Synopsis</b>	rmonRet __stdcall rmonObjectReadX(HRMONCON hCon, const int table, const char* name, rmoncdobjectx output, void* uptr)																																								
<b>Description</b>	Fetches objects from the device.																																								
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>table</b></td> <td>The object table to use</td> </tr> <tr> <td><b>name</b></td> <td>The name of the object. Wildcards can be used.</td> </tr> <tr> <td><b>output</b></td> <td>Pointer to callback function which is called for each object read</td> </tr> <tr> <td><b>uptr</b></td> <td>Pointer to user argument used in callback function</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>table</b>	The object table to use	<b>name</b>	The name of the object. Wildcards can be used.	<b>output</b>	Pointer to callback function which is called for each object read	<b>uptr</b>	Pointer to user argument used in callback function																													
<b>hCon</b>	Handle to connection																																								
<b>table</b>	The object table to use																																								
<b>name</b>	The name of the object. Wildcards can be used.																																								
<b>output</b>	Pointer to callback function which is called for each object read																																								
<b>uptr</b>	Pointer to user argument used in callback function																																								
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonError, rmonNoData, rmonCancelled																																								
	<p>The call-back function is defined as follows:</p> <pre>typedef int (__stdcall *rmoncbobjectx)(void* uptr, const char* name, int type, int size, unsigned char* data, unsigned long flags, int min_size, unsigned char* min_val, int max_size, unsigned char* max_val, const char* description);</pre> <p>The callback must return 1 to continue to the next object. Return 0 to stop fetching the objects.</p>																																								
	<table border="1"> <tr> <td><b>uptr</b></td> <td colspan="2">User pointer</td> </tr> <tr> <td><b>name</b></td> <td colspan="2">The name of the object</td> </tr> <tr> <td><b>type</b></td> <td colspan="2">The type of the object. (See object types above)</td> </tr> <tr> <td><b>size</b></td> <td colspan="2">The size of the data in bytes</td> </tr> <tr> <td><b>data</b></td> <td colspan="2">The object data.</td> </tr> <tr> <td><b>flags</b></td> <td colspan="2">Object flags:</td> </tr> <tr> <td></td> <td><b>0x0001</b></td> <td>Read only object</td> </tr> <tr> <td></td> <td><b>0x0002</b></td> <td>Encrypted object</td> </tr> <tr> <td><b>min_size</b></td> <td colspan="2">The size of the min_val parameter in bytes.</td> </tr> <tr> <td><b>min_val</b></td> <td colspan="2">Pointer to a buffer, containing the minimum value or the minimum length, of the object. The format of the buffer depends on the object type.</td> </tr> <tr> <td><b>max_size</b></td> <td colspan="2">The size of the max_val parameter in bytes.</td> </tr> <tr> <td><b>max_val</b></td> <td colspan="2">Pointer to a buffer, containing the maximum value or the maximum length, of the object. The format of the buffer depends on the object type.</td> </tr> <tr> <td><b>description</b></td> <td colspan="2">A short description of the object.</td> </tr> </table>		<b>uptr</b>	User pointer		<b>name</b>	The name of the object		<b>type</b>	The type of the object. (See object types above)		<b>size</b>	The size of the data in bytes		<b>data</b>	The object data.		<b>flags</b>	Object flags:			<b>0x0001</b>	Read only object		<b>0x0002</b>	Encrypted object	<b>min_size</b>	The size of the min_val parameter in bytes.		<b>min_val</b>	Pointer to a buffer, containing the minimum value or the minimum length, of the object. The format of the buffer depends on the object type.		<b>max_size</b>	The size of the max_val parameter in bytes.		<b>max_val</b>	Pointer to a buffer, containing the maximum value or the maximum length, of the object. The format of the buffer depends on the object type.		<b>description</b>	A short description of the object.	
<b>uptr</b>	User pointer																																								
<b>name</b>	The name of the object																																								
<b>type</b>	The type of the object. (See object types above)																																								
<b>size</b>	The size of the data in bytes																																								
<b>data</b>	The object data.																																								
<b>flags</b>	Object flags:																																								
	<b>0x0001</b>	Read only object																																							
	<b>0x0002</b>	Encrypted object																																							
<b>min_size</b>	The size of the min_val parameter in bytes.																																								
<b>min_val</b>	Pointer to a buffer, containing the minimum value or the minimum length, of the object. The format of the buffer depends on the object type.																																								
<b>max_size</b>	The size of the max_val parameter in bytes.																																								
<b>max_val</b>	Pointer to a buffer, containing the maximum value or the maximum length, of the object. The format of the buffer depends on the object type.																																								
<b>description</b>	A short description of the object.																																								

<b>rmonObjectWrite()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State	
<b>Synopsis</b>	rmonRet __stdcall rmonObjectWrite(HRMONCON hCon, const int table, const short flags, const int count, rmonObjectInfo* data, int* index)		
<b>Description</b>	Write objects to the device.		
<b>Input</b>	<b>hCon</b>	<b>Handle to connection</b>	
	<b>table</b>	The object table to use	
	<b>flags</b>	Flags to control the write operation. See below.	
	<b>count</b>	The number of objects in the array	
	<b>data</b>	Pointer to an array of objects to write	
<b>Output</b>	<b>index</b>	The index of the object which failed. 0 if the error is not from an object or if there is no error	
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonError, rmonNoData		
	<pre>typedef struct {     char          name[255];     int           type;     int           size;     unsigned char* data;     unsigned long flags;     int           min_val;     int           max_val;     char          desc[81]; } rmonObjectInfo;</pre>		
	Object flags:		
	<b>0x0001</b>	Read only object	
	<b>0x0002</b>	Encrypted object	
	Write flags:		
	<b>Symbolic name</b>	<b>Value</b>	<b>Description</b>
	<b>RMONOBJ_FLAG_OVERWRITE</b>	0x0001	Overwrite any existing objects
	<b>RMONOBJ_FLAG_IGNORE</b>	0x0002	Only update existing objects.

<b>rmonObjectWriteX()</b>		<b>RTC architecture:</b> NX32L <b>Called in:</b> Connected State	
<b>Synopsis</b>	rmonRet __stdcall rmonObjectWriteX(HRMONCON hCon, const int table, const short flags, const int count, rmonObjectInfoX* data, int* index)		
<b>Description</b>	Write objects to the device.		
<b>Input</b>	<b>hCon</b>	Handle to connection	
	<b>table</b>	The object table to use	
	<b>flags</b>	Flags to control the write operation. See below.	
	<b>count</b>	The number of objects in the array	
	<b>data</b>	Pointer to an array of objects to write	
<b>Output</b>	<b>index</b>	The index of the object which failed. 0 if the error is not from an object or if there is no error	
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonError, rmonNoData		
	<pre>typedef struct {     char          name[255];     int           type;     int           size;     unsigned char* data;     unsigned long flags;     int           min_size;     unsigned char* min_val;     int           max_size;     unsigned char* max_val;     char          desc[81]; } rmonObjectInfoX;</pre>		
	Object flags:		
	<b>0x0001</b>	Read only object	
	<b>0x0002</b>	Encrypted object	
	Write flags:		
	<b>Symbolic name</b>	<b>Value</b>	<b>Description</b>
	<b>RMONOBJ_FLAG_OVERWRITE</b>	0x0001	Overwrite any existing objects
	<b>RMONOBJ_FLAG_IGNORE</b>	0x0002	Only update existing objects.

<b>rmonObjectErase()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonObjectErase(HRMONCON hCon, const int table, const char* name)	
<b>Description</b>	Erase an object from the device.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>table</b>	The object table to use
	<b>name</b>	The name of the object. Wildcards can be used.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonError	

## Misc. functions

Below is a list of different “housekeeping” functions.

<b>rmonReset()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonReset(HRMONCON hCon)	
<b>Description</b>	<p>This function will reset the connected RTCU device. If the RTCU device is remotely connected (RCH) the reset will be delayed until the connection is lost (by calling rmonDisconnect() etc). However, if the connection is through a direct cable connection, the RTCU device executes the reset command immediately. The reset has the same effect as cycling power to the RTCU device, the VPL program starts executing from the start again. This can also be carried out from the RTCU M2M Studio (menu: Device -&gt; Execution -&gt; Reset)</p> <p>(This function does the same as the VPL function boardReset()).</p>	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle	

<b>rmonResetX()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonReset(HRMONCON hCon, int level)	
<b>Description</b>	<p>This function will reset the connected RTCU device. If the RTCU device is remotely connected (RCH) the reset will be delayed until the connection is lost (by calling rmonDisconnect() etc). However, if the connection is through a direct cable connection, the RTCU device executes the reset command immediately.</p> <p>For remote connections, a system reset might halt the application immediately and then reboot when the connection is lost.</p> <p>This can also be carried out from the RTCU M2M Studio (menu: Device -&gt; Execution -&gt; Reset and Device -&gt; Execution -&gt; System Reboot)</p> <p>(This function does the same as the VPL function boardReset()).</p>	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>level</b>	The type of reset to perform. 0 = Normal reset, the same as rmonReset(). 1 = System reset (NX32L only).
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonIllegalTarget, rmonError	

<b>rmonHalt()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonHalt(HRMONCON hCon)	
<b>Description</b>	<p>Stops the currently executing VPL program in the RTCU.</p> <p>This can also be carried out from the RTCU M2M Studio (menu: Device -&gt; Execution -&gt; Halt)</p> <p>The RTCU device can be started again with the reset command (see above) or by cycling power.</p>	

<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonGetSerialNumber()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonGetSerialNumber(HRMONCON hCon, unsigned long *SerialNumber)	
<b>Description</b>	Returns the serial number of the connected RTCU.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>SerialNumber</b>	The serialnumber of the RTCU device.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonIllegalTarget	

<b>rmonVer()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State						
<b>Synopsis</b>	rmonRet __stdcall rmonVer(HRMONCON hCon, int *ver)							
<b>Description</b>	Returns the Firmware version of the connected RTCU. On NX32L devices, the major and minor part of the runtime version is returned.  Note that this function must be used to determine if RTCU device is in monitor mode.							
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> </table>		<b>hCon</b>	Handle to connection				
<b>hCon</b>	Handle to connection							
<b>Output</b>	<table border="1"> <tr> <td><b>ver</b></td> <td>           Firmware version, always different from 0 and scaled by 100 (Version 4.66 is returned as 466). Note that a version higher than 90.00 means that the RTCU is in monitor mode.             On NX32L devices, the major and minor part of the runtime version is returned as a binary packed value using this format:           <table border="1"> <tr> <td><b>major</b></td> <td>bit 8..15</td> </tr> <tr> <td><b>minor</b></td> <td>bit 0..7</td> </tr> </table>           A decimal value larger than 9000 means that the RTCU is in monitor mode.         </td> </tr> </table>		<b>ver</b>	Firmware version, always different from 0 and scaled by 100 (Version 4.66 is returned as 466). Note that a version higher than 90.00 means that the RTCU is in monitor mode.  On NX32L devices, the major and minor part of the runtime version is returned as a binary packed value using this format: <table border="1"> <tr> <td><b>major</b></td> <td>bit 8..15</td> </tr> <tr> <td><b>minor</b></td> <td>bit 0..7</td> </tr> </table> A decimal value larger than 9000 means that the RTCU is in monitor mode.	<b>major</b>	bit 8..15	<b>minor</b>	bit 0..7
<b>ver</b>	Firmware version, always different from 0 and scaled by 100 (Version 4.66 is returned as 466). Note that a version higher than 90.00 means that the RTCU is in monitor mode.  On NX32L devices, the major and minor part of the runtime version is returned as a binary packed value using this format: <table border="1"> <tr> <td><b>major</b></td> <td>bit 8..15</td> </tr> <tr> <td><b>minor</b></td> <td>bit 0..7</td> </tr> </table> A decimal value larger than 9000 means that the RTCU is in monitor mode.	<b>major</b>	bit 8..15	<b>minor</b>	bit 0..7			
<b>major</b>	bit 8..15							
<b>minor</b>	bit 0..7							
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle							

<b>rmonSetPassword()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State				
<b>Synopsis</b>	rmonRet __stdcall rmonSetPassword (HRMONCON hCon, const char password[21])					
<b>Description</b>	Sets new password for access to RTCU. This is the password that is to be used in rmonAuthenticate(). An empty string will disable password protection. This can also be set from the RTCU M2M Studio (menu: Device -> Configuration -> Set Password)					
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>password</b></td> <td>New password to be set (zero terminated ASCII String).</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>password</b>	New password to be set (zero terminated ASCII String).
<b>hCon</b>	Handle to connection					
<b>password</b>	New password to be set (zero terminated ASCII String).					
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError					

<b>rmonGetTargetInfo()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonGetTargetInfo(HRMONCON hCon, int* targetID, int* firmwareVer);	
<b>Description</b>	Fetches RTCU type and firmware version from the RTCU. On NX32L devices, the runtime version is returned. (see <a href="#">rmonGetDeviceInfo()</a> )	
<b>Input</b>	<b>hCon</b>	<b>Handle to connection</b>
<b>Output</b>	<b>targetID</b>	Please see the table below for a list of possible target ID's.
	<b>firmwareVer</b>	Firmware version always different from 0 and scaled by 100 (Version 4.66 is returned as 466) NX32L devices use a different format, see <a href="#">rmonGetDeviceInfo()</a>
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle	
	<b>Symbolic name</b>	<b>Value</b> <b>Description</b>
	RMONTGT_ICP002	1      RTCU SA
	RMONTGT_ICP003	2      RTCU DIN
	RMONTGT_ICP004	4      RTCU D4
	RMONTGT_ICP005	5      RTCU A5 / RTCU A5i
	RMONTGT_ICP006	6      RTCU A6
	RMONTGT_ICP007	7      RTCU M7
	RMONTGT_ICP009	9      RTCU A9i
	RMONTGT_ICP010	10     RTCU M10 Series
	RMONTGT_ICP011	11     RTCU M11 Series
	RMONTGT_ICP102	102    RTCU MX2 Series
	RMONTGT_ICP103	103    RTCU DX4i mk2
	RMONTGT_ICP104	104    RTCU DX4 Series
	RMONTGT_ICP105	105    RTCU CX1 Series
	RMONTGT_ICP106	106    RTCU SX1 Series
	RMONTGT_ICP109	109    RTCU AX9 Series
	RMONTGT_ICP122	122    RTCU MX2 turbo Series
	RMONTGT_ICP129	129    RTCU AX9 turbo / AX9 encore Series
	RMONTGT_ICP192	192    RTCU MX2 SOM
	RMONTGT_ICP202	202    RTCU NX-200 Series
	RMONTGT_ICP204	204    RTCU NX-400
	RMONTGT_ICP209	209    RTCU NX-900 Series
	RMONTGT_ICP302	302    RTCU LX2 Series
	RMONTGT_ICP304	304    RTCU LX4 Series
	RMONTGT_ICP305	305    RTCU LX5 Series
	RMONTGT_ICP309	309    RTCU LX9 pro
	RMONTGT_ICP315	315    RTCU LX5 ultra
	RMONTGT_ICP316	316    RTCU LX5 core
	RMONTGT_ICP319	319    RTCU LX9 eco

<b>rmonGetTargetProfile()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonGetTargetProfile(HRMONCON hCon, int* targetID, int* firmwareVer, unsigned long* SerialNumber);	
<b>Description</b>	Fetches RTCU type, serial number and firmware version from the RTCU. On NX32L devices, the runtime version is returned. (see <a href="#">rmonGetDeviceInfo()</a> )	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>targetID</b>	Please see the table in rmonGetTargetInfo for a list of target ID's.
	<b>firmwareVer</b>	Firmware version always different from 0 and scaled by 100 (Version 4.66 is returned as 466) NX32L devices use a different format, see <a href="#">rmonGetDeviceInfo()</a>
	<b>SerialNumber</b>	The serialnumber of the RTCU device.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle	

<b>rmonGetDeviceInfo()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State								
<b>Synopsis</b>	rmonRet __stdcall rmonGetDeviceInfo(HRMONCON hCon, int* targetID, int* VerRuntime, int* VerSystem, unsigned long* SerialNumber);									
<b>Description</b>	Fetches RTCU type, serial number and firmware version from the RTCU.  On NX32L devices, the version is <major>.<minor>.<build>; which is returned as a binary packed value using this format:									
	<table border="1"> <tr> <td><b>reserved</b></td> <td>bit 24..31</td> </tr> <tr> <td><b>major</b></td> <td>bit 16..23</td> </tr> <tr> <td><b>minor</b></td> <td>bit 8..15</td> </tr> <tr> <td><b>build</b></td> <td>bit 0..7</td> </tr> </table>		<b>reserved</b>	bit 24..31	<b>major</b>	bit 16..23	<b>minor</b>	bit 8..15	<b>build</b>	bit 0..7
<b>reserved</b>	bit 24..31									
<b>major</b>	bit 16..23									
<b>minor</b>	bit 8..15									
<b>build</b>	bit 0..7									
	On other RTCU devices the version is scaled by 100. (Version 4.66 is returned as 466)									
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> </table>		<b>hCon</b>	Handle to connection						
<b>hCon</b>	Handle to connection									
<b>Output</b>	<table border="1"> <tr> <td><b>targetID</b></td> <td>Please see the table in rmonGetTargetInfo for a list of target ID's.</td> </tr> <tr> <td><b>VerRuntime</b></td> <td>The version of the runtime on the RTCU. This is always different from 0.</td> </tr> <tr> <td><b>VerSystem</b></td> <td>The version of the system on the RTCU. This will be 0 on RTCU without a system version</td> </tr> <tr> <td><b>SerialNumber</b></td> <td>The serialnumber of the RTCU device.</td> </tr> </table>		<b>targetID</b>	Please see the table in rmonGetTargetInfo for a list of target ID's.	<b>VerRuntime</b>	The version of the runtime on the RTCU. This is always different from 0.	<b>VerSystem</b>	The version of the system on the RTCU. This will be 0 on RTCU without a system version	<b>SerialNumber</b>	The serialnumber of the RTCU device.
<b>targetID</b>	Please see the table in rmonGetTargetInfo for a list of target ID's.									
<b>VerRuntime</b>	The version of the runtime on the RTCU. This is always different from 0.									
<b>VerSystem</b>	The version of the system on the RTCU. This will be 0 on RTCU without a system version									
<b>SerialNumber</b>	The serialnumber of the RTCU device.									
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle									

<b>rmonReceiveDebugMsg()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonReceiveDebugMsg (HRMONCON hCon, char* msg, int maxsize)	
<b>Description</b>	Receive any incoming Debug messages from RTCU. Please notice that rmonReceiveDebugMsg blocks and will not return before a debug message has been received.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>maxsize</b>	Maximum number of characters to receive
<b>Output</b>	<b>msg</b>	Buffer with received debug message
<b>Returns</b>	rmonOK, rmonIllegalHandle, rmonNoData	

<b>rmonGetDebugEnable()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonGetDebugEnable(HRMONCON hCon, int* enabled )	
<b>Description</b>	Checks if Debug messages have been enabled or disabled in device.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	enabled	1 if Debug messages are enabled, 0 if Debug messages are disabled
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonSetDebugEnable()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonSetDebugEnable(HRMONCON hCon, int enabled )	
<b>Description</b>	Enable or disable Debug messages from device.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>enabled</b>	1 to enable Debug messages, 0 to disable Debug messages
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonVoiceMessagesAbove64K()</b>		<b>RTCU architecture:</b> X32 <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonVoiceMessagesAbove64K (HRMONCON hCon, char *above)	
<b>Description</b>	Determine if voice messages are stored above 64k. This function is used to determine if voice messages will be overwritten by the rmonApplicationStartUpload function or the rmonFirmwareStartUpload function.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>above</b>	1 if Voice messages above 64k, 0 if no Voice messages above 64k
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonGetAppInfo()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonGetAppInfo(HRMONCON hCon, char *Appname, int *Appver )	
<b>Description</b>	Fetches Application name and version from the RTCU.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>Appname</b>	0 (zero) terminated string containing the application name. Max. 15 characters long.
	<b>Appver</b>	Application version scaled by 100 (Version 4.66 is returned as 466)
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData	

<b>rmonGetGPRSSettings()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State								
<b>Synopsis</b>	rmonRet __stdcall rmonGetGPRSSettings(HRMONCON hCon, rmonGPRSSettings* Settings);									
<b>Description</b>	<p>This function fetches the TCP/IP settings the device uses to connect over GPRS. The GPRS settings retrieved from the RTCU device are identical to those retrieved with the "Fetch" button in the RTCU M2M Studio (Device-&gt;Network-&gt;Network settings).</p> <p>All the general TCP/IP parameters use a binary packed IP address (a.b.c.d) using this format:</p> <table border="1" data-bbox="347 1122 655 1252"> <tr> <td><b>a</b></td> <td>bit 24..31</td> </tr> <tr> <td><b>b</b></td> <td>bit 16..23</td> </tr> <tr> <td><b>c</b></td> <td>bit 8..15</td> </tr> <tr> <td><b>d</b></td> <td>bit 0..7</td> </tr> </table>		<b>a</b>	bit 24..31	<b>b</b>	bit 16..23	<b>c</b>	bit 8..15	<b>d</b>	bit 0..7
<b>a</b>	bit 24..31									
<b>b</b>	bit 16..23									
<b>c</b>	bit 8..15									
<b>d</b>	bit 0..7									
<b>Input</b>	<b>hCon</b>	Handle to connection								
<b>Output</b>	<b>Settings</b>	A structure containing the GPRS settings								
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData									
	<pre>typedef struct {     // general TCP/IP parameters:     unsigned long ip_address;     unsigned long subnet_mask;     unsigned long gateway;     unsigned long dns_1;     unsigned long dns_2;     // PPP parameters:     char username[34];     char password[34];     // Dialup/GPRS parameters:     char APN[34];     unsigned short authentication; } rmonGPRSSettings;</pre>									

<b>rmonSetGPRSSettings()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State								
<b>Synopsis</b>	rmonRet __stdcall rmonSetGPRSSettings(HRMONCON hCon, rmonGPRSSettings Settings);									
<b>Description</b>	<p>This function sets the TCP/IP settings the device uses to connect over GPRS. This function is identical to the VPL function netSetMobileParam, and the TCP/IP settings dialog (Device-&gt;Network-&gt;Network settings) in the RTCU M2M Studio.</p> <p>All the general TCP/IP parameters use a binary packed IP address (a.b.c.d) using this format:</p> <table border="1"> <tr> <td><b>a</b></td> <td>bit 24..31</td> </tr> <tr> <td><b>b</b></td> <td>bit 16..23</td> </tr> <tr> <td><b>c</b></td> <td>bit 8..15</td> </tr> <tr> <td><b>d</b></td> <td>bit 0..7</td> </tr> </table>		<b>a</b>	bit 24..31	<b>b</b>	bit 16..23	<b>c</b>	bit 8..15	<b>d</b>	bit 0..7
<b>a</b>	bit 24..31									
<b>b</b>	bit 16..23									
<b>c</b>	bit 8..15									
<b>d</b>	bit 0..7									
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>Settings</b></td> <td>A structure containing the GPRS settings</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>Settings</b>	A structure containing the GPRS settings				
<b>hCon</b>	Handle to connection									
<b>Settings</b>	A structure containing the GPRS settings									
<b>Returns</b>	<p>rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError</p> <pre>typedef struct {     // general TCP/IP parameters:     unsigned long ip_address;     unsigned long subnet_mask;     unsigned long gateway;     unsigned long dns_1;     unsigned long dns_2;     // PPP parameters:     char username[34];     char password[34];     // Dialup/GPRS parameters:     char APN[34];     unsigned short authentication; } rmonGPRSSettings;</pre>									

<b>rmonGetGatewaySettings()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonGetGatewaySettings(HRMONCON hCon, rmonGWSettings* Settings);	
<b>Description</b>	This function fetches the settings the device uses to connect to the RCH. The RCH settings retrieved from the RTCU device are identical to those retrieved in the RTCU M2M Studio with the "Fetch" button in the RCH settings dialog (Device->Network->RTCU Communication Hub settings).	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>Settings</b>	A structure containing the RCH settings
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	
	<pre> typedef struct {     // RTCU Communication Hub parameters:     unsigned short gw_enabled;     char gw_ip[42];     unsigned short gw_port;     char gw_key[10];     char phonenumber_sms[22];     unsigned char crypt_key[16];     // advanced settings (modification not recommended):     unsigned short max_connection_attempt;     unsigned short max_send_req_attempt;     unsigned short response_timeout;     unsigned short alive_freq; } rmonGWSettings; </pre>	

<b>rmonSetGatewaySettings()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State				
<b>Synopsis</b>	rmonRet __stdcall rmonSetGatewaySettings(HRMONCON hCon, rmonGWSettings Settings);					
<b>Description</b>	This function fetches the settings the device uses to connect to RCH. This function is identical to the VPL functions sockSetGWParam and rchFallbackSet, and the settings are identical to those written from the RTCU M2M Studio with the "Apply" button in the RTCU Communication Hub settings dialog (Device->Network->RTCU Communication Hub settings).					
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>Settings</b></td> <td>A structure containing the Gateway settings</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>Settings</b>	A structure containing the Gateway settings
<b>hCon</b>	Handle to connection					
<b>Settings</b>	A structure containing the Gateway settings					
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError					
	<pre>typedef struct {     // RTCU Communication Hub parameters:     unsigned short gw_enabled;     char gw_ip[42];     unsigned short gw_port;     char gw_key[10];     char phonenumber_sms[22];     unsigned char crypt_key[16];     // advanced settings (modification not recommended):     unsigned short max_connection_attempt;     unsigned short max_send_req_attempt;     unsigned short response_timeout;     unsigned short alive_freq; } rmonGWSettings;</pre>					

<b>rmonRCHGetConfig()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonRCHGetConfig(HRMONCON hCon, const int index, rmonRCHConfig* Settings);	
<b>Description</b>	<p>This function fetches the RTCU Communication Hub settings the device uses to connect to server.</p> <p>The RCH settings retrieved from the RTCU device are identical to those retrieved in the RTCU M2M Studio with the "Fetch" button in the RTCU Communication Hub settings dialog (Device→Network→RTCU Communication Hub settings).</p>	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>index</b>	The index of the configuration to read. 0 = Fallback config, 1 = Primary server config.
<b>Output</b>	<b>Settings</b>	A structure containing the settings
<b>Returns</b>	<p>rmonOK, rmonErrorr, rmonComError, rmonIllegalHandle, rmonTargetError</p> <pre> struct rmonRCHConfig {     unsigned long    size;     unsigned char    enabled;     unsigned char    secure;     char             host[51];     unsigned char    iface;     unsigned char    crypt_enable;     unsigned short   port;     char             login_key[10];     unsigned char    crypt_key[16];     unsigned short   max_connection_attempt;     unsigned short   max_send_req_attempt;     unsigned short   response_timeout;     unsigned short   alive_freq; }; </pre>	

<b>rmonRCHSetConfig()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State						
<b>Synopsis</b>	rmonRet __stdcall rmonRCHSetConfig(HRMONCON hCon, const int index, rmonGWSettings Settings);							
<b>Description</b>	<p>This function fetches the RTCU Communication Hub settings the device uses to connect to the server.</p> <p>This function is identical to the VPL function rchConfigSet, and the settings are identical to those written from the RTCU M2M Studio with the “Apply” button in the RTCU Communication Hub settings dialog (Device→Network→RTCU Communication Hub settings).</p>							
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>index</b></td> <td>The index of the configuration to read. 0 = Fallback config, 1 = Primary server config.</td> </tr> <tr> <td><b>Settings</b></td> <td>A structure containing the RCH settings</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>index</b>	The index of the configuration to read. 0 = Fallback config, 1 = Primary server config.	<b>Settings</b>	A structure containing the RCH settings
<b>hCon</b>	Handle to connection							
<b>index</b>	The index of the configuration to read. 0 = Fallback config, 1 = Primary server config.							
<b>Settings</b>	A structure containing the RCH settings							
<b>Returns</b>	<p>rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError</p> <pre> struct rmonGWSettings {     unsigned long    size;     unsigned char    enabled;     unsigned char    secure;     char             host[51];     unsigned char    iface;     unsigned char    crypt_enable;     unsigned short   port;     char            login_key[10];     unsigned char    crypt_key[16];     unsigned short   max_connection_attempt;     unsigned short   max_send_req_attempt;     unsigned short   response_timeout;     unsigned short   alive_freq; }; </pre>							

<b>rmonRCHGetAutoConnect()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State		
<b>Synopsis</b>	rmonRet __stdcall rmonRCHGetAutoConnect(HRMONCON hCon, int *enable)			
<b>Description</b>	This function fetches whether the RTCU device will start the connection to the RTCU Communication Hub automatically, or requires the application to do this manually.			
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> </table>		<b>hCon</b>	Handle to connection
<b>hCon</b>	Handle to connection			
<b>Output</b>	<table border="1"> <tr> <td><b>enable</b></td> <td>1 if auto connect is enabled, 0 if auto connect is disabled</td> </tr> </table>		<b>enable</b>	1 if auto connect is enabled, 0 if auto connect is disabled
<b>enable</b>	1 if auto connect is enabled, 0 if auto connect is disabled			
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData			

<b>rmonRCHSetAutoConnect()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State				
<b>Synopsis</b>	rmonRet __stdcall rmonRCHGetAutoConnect(HRMONCON hCon, int *enable)					
<b>Description</b>	This function sets whether the RTCU device will start the connection to the RTCU Communication Hub automatically, or requires the application to do this manually.					
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>enable</b></td> <td>1 if auto connect is enabled, 0 if auto connect is disabled</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>enable</b>	1 if auto connect is enabled, 0 if auto connect is disabled
<b>hCon</b>	Handle to connection					
<b>enable</b>	1 if auto connect is enabled, 0 if auto connect is disabled					
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData					

<b>rmonGetLANSettings()</b>		<b>RTCU architecture:</b> NX32 & NX32L <b>Called in:</b> Connected State									
<b>Synopsis</b>	rmonRet __stdcall rmonGetLANSettings(HRMONCON hCon, int iface, rmonNetwork* Settings);										
<b>Description</b>	This function fetches the settings the device uses to connect over Ethernet. The LAN settings retrieved from the RTCU device are identical to those retrieved in the RTCU M2M Studio with the "Fetch" button in the Network settings dialog (Device->Network->Network settings).										
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>iface</b></td> <td>The network interface</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>iface</b>	The network interface					
<b>hCon</b>	Handle to connection										
<b>iface</b>	The network interface										
<b>Output</b>	<table border="1"> <tr> <td><b>Settings</b></td> <td>A structure containing the LAN settings</td> </tr> </table>		<b>Settings</b>	A structure containing the LAN settings							
<b>Settings</b>	A structure containing the LAN settings										
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError										
	<pre>typedef struct {     unsigned short flags;     unsigned long  addr;     unsigned long  subnet;     unsigned long  gateway;     unsigned long  dns_1;     unsigned long  dns_2; } rmonNetwork;</pre>										
	Available flags:										
	<table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td><b>RMON_NET_DHCP</b></td> <td>0x0001</td> <td>Use DHCP to get TCP/IP address.</td> </tr> <tr> <td><b>RMON_NET_AUTODNS</b></td> <td>0x0002</td> <td>Use DNS servers from DHCP lookup.</td> </tr> </tbody> </table>		Symbolic name	Value	Description	<b>RMON_NET_DHCP</b>	0x0001	Use DHCP to get TCP/IP address.	<b>RMON_NET_AUTODNS</b>	0x0002	Use DNS servers from DHCP lookup.
Symbolic name	Value	Description									
<b>RMON_NET_DHCP</b>	0x0001	Use DHCP to get TCP/IP address.									
<b>RMON_NET_AUTODNS</b>	0x0002	Use DNS servers from DHCP lookup.									

<b>rmonSetLANSettings()</b>		<b>RTCU architecture:</b> NX32 & NX32L <b>Called in:</b> Connected State									
<b>Synopsis</b>	rmonRet __stdcall rmonSetLANSettings(HRMONCON hCon, int iface, rmonNetwork Settings);										
<b>Description</b>	This function sets the settings the device uses to connect over Ethernet. This function is identical to the VPL function netSetLANParam, and the settings are identical to those written from the RTCU M2M Studio with the "Apply" button in the Network settings dialog (Device->Network->Network settings).										
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>iface</b></td> <td>The network interface</td> </tr> <tr> <td><b>Settings</b></td> <td>A structure containing the LAN settings</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>iface</b>	The network interface	<b>Settings</b>	A structure containing the LAN settings			
<b>hCon</b>	Handle to connection										
<b>iface</b>	The network interface										
<b>Settings</b>	A structure containing the LAN settings										
<b>Output</b>											
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError										
	<pre>typedef struct {     unsigned short flags;     unsigned long  addr;     unsigned long  subnet;     unsigned long  gateway;     unsigned long  dns_1;     unsigned long  dns_2; } rmonNetwork;</pre>										
	<b>Available flags:</b> <table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>RMON_NET_DHCP</td> <td>0x0001</td> <td>Use DHCP to get TCP/IP address.</td> </tr> <tr> <td>RMON_NET_AUTODNS</td> <td>0x0002</td> <td>Use DNS servers from DHCP lookup.</td> </tr> </tbody> </table>		Symbolic name	Value	Description	RMON_NET_DHCP	0x0001	Use DHCP to get TCP/IP address.	RMON_NET_AUTODNS	0x0002	Use DNS servers from DHCP lookup.
Symbolic name	Value	Description									
RMON_NET_DHCP	0x0001	Use DHCP to get TCP/IP address.									
RMON_NET_AUTODNS	0x0002	Use DNS servers from DHCP lookup.									

<b>rmonGetWLANSettings()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State									
<b>Synopsis</b>	rmonRet __stdcall rmonGetWLANSettings(HRMONCON hCon, int index, rmonWLANSettings* Settings);										
<b>Description</b>	This function fetches the settings the device uses to connect over WiFi. The WLAN settings retrieved from the RTCU device are identical to those retrieved in the RTCU M2M Studio with the "Fetch" button in the Network settings dialog (Device->Network->Network settings).										
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>index</b></td> <td>The index of the wireless network information</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>index</b>	The index of the wireless network information					
<b>hCon</b>	Handle to connection										
<b>index</b>	The index of the wireless network information										
<b>Output</b>	<table border="1"> <tr> <td><b>Settings</b></td> <td>A structure containing the WLAN settings</td> </tr> </table>		<b>Settings</b>	A structure containing the WLAN settings							
<b>Settings</b>	A structure containing the WLAN settings										
<b>Returns</b>	rmonOK, rmonComError, rmonError, rmonIllegalHandle, rmonTargetError  <pre> typedef struct {     unsigned short flags;     unsigned long  addr;     unsigned long  subnet;     unsigned long  gateway;     unsigned long  dns_1;     unsigned long  dns_2; } rmonNetwork;  typedef struct {     char          phrase[64]; } rmonWLANSecurityWPA;  typedef struct {     unsigned char ssid[32];     unsigned short security;     union {         rmonWLANSecurityWPA wpa;     } sec;     rmonNetwork  tcpip; } rmonWLANSettings;           </pre> <p>Available flags:</p> <table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td><b>RMON_NET_DHCP</b></td> <td>0x0001</td> <td>Use DHCP to get TCP/IP address.</td> </tr> <tr> <td><b>RMON_NET_AUTODNS</b></td> <td>0x0002</td> <td>Use DNS servers from DHCP lookup.</td> </tr> </tbody> </table>		Symbolic name	Value	Description	<b>RMON_NET_DHCP</b>	0x0001	Use DHCP to get TCP/IP address.	<b>RMON_NET_AUTODNS</b>	0x0002	Use DNS servers from DHCP lookup.
Symbolic name	Value	Description									
<b>RMON_NET_DHCP</b>	0x0001	Use DHCP to get TCP/IP address.									
<b>RMON_NET_AUTODNS</b>	0x0002	Use DNS servers from DHCP lookup.									

<b>rmonSetWLANSettings()</b>		<b>RTCU architecture:</b> NX32L <b>Called in:</b> Connected State									
<b>Synopsis</b>	rmonRet __stdcall rmonSetWLANSettings(HRMONCON hCon, int index, rmonWLANSettings Settings);										
<b>Description</b>	This function sets the settings the device uses to connect over WiFi. This function is identical to the VPL function netSetWLANParam, and the settings are identical to those written from the RTCU M2M Studio with the "Apply" button in the Network settings dialog (Device->Network->Network settings).										
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>index</b></td> <td>The index of the wireless network information</td> </tr> <tr> <td><b>Settings</b></td> <td>A structure containing the WLAN settings</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>index</b>	The index of the wireless network information	<b>Settings</b>	A structure containing the WLAN settings			
<b>hCon</b>	Handle to connection										
<b>index</b>	The index of the wireless network information										
<b>Settings</b>	A structure containing the WLAN settings										
<b>Returns</b>	rmonOK, rmonComError, rmonError, rmonIllegalHandle, rmonTargetError  <pre> typedef struct {     unsigned short flags;     unsigned long  addr;     unsigned long  subnet;     unsigned long  gateway;     unsigned long  dns_1;     unsigned long  dns_2; } rmonNetwork;  typedef struct {     char          phrase[64]; } rmonWLANSecurityWPA;  typedef struct {     unsigned char  ssid[32];     unsigned short security;     union {         rmonWLANSecurityWPA wpa;     } sec;     rmonNetwork      tcpip; } rmonWLANSettings;           </pre> <p>Available flags:</p> <table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td><b>RMON_NET_DHCP</b></td> <td>0x0001</td> <td>Use DHCP to get TCP/IP address.</td> </tr> <tr> <td><b>RMON_NET_AUTODNS</b></td> <td>0x0002</td> <td>Use DNS servers from DHCP lookup.</td> </tr> </tbody> </table>		Symbolic name	Value	Description	<b>RMON_NET_DHCP</b>	0x0001	Use DHCP to get TCP/IP address.	<b>RMON_NET_AUTODNS</b>	0x0002	Use DNS servers from DHCP lookup.
Symbolic name	Value	Description									
<b>RMON_NET_DHCP</b>	0x0001	Use DHCP to get TCP/IP address.									
<b>RMON_NET_AUTODNS</b>	0x0002	Use DNS servers from DHCP lookup.									

<b>rmonFaultLogRead()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFaultLogRead(HRMONCON hCon, rmonFault *Fault)	
<b>Description</b>	Fetches the Fault log from the RTCU. This function is identical to the fetch button in the fault log in the RTCU M2M Studio	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>Fault</b>	The function fills this structure with the fault log entries.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	
	<pre> typedef struct {     unsigned short    year;     unsigned char     month;     unsigned char     date;     unsigned char     hour;     unsigned char     minute;     unsigned char     second;     unsigned char     Code; } rmonFaultRecord;  typedef struct {     unsigned char     NumRecords;     unsigned char     NextIn; // Index where the next record will be inserted     rmonFaultRecord  Record[32]; } rmonFault; </pre>	

<b>rmonFaultLogReadX()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFaultLogRead(HRMONCON hCon, rmonFaultX *Fault)	
<b>Description</b>	Fetches the Fault log from the RTCU, with debug information. This function is identical to the fetch button in the fault log in the RTCU M2M Studio	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>Fault</b>	The function fills this structure with the fault log entries.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	
	<pre> typedef struct {     unsigned short    year;     unsigned char     month;     unsigned char     date;     unsigned char     hour;     unsigned char     minute;     unsigned char     second;     unsigned char     code;     char              filename[16];     unsigned short    line;     unsigned short    threadid;     unsigned char     data[38]; } rmonFaultRecordX;  typedef struct {     unsigned char     NumRecords;     unsigned char     NextIn; // Index where the next record will be inserted     rmonFaultRecordX Record[16]; } rmonFaultX; </pre>	

<b>rmonFaultLogClear()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFaultLogClear(HRMONCON hCon)	
<b>Description</b>	This function clears the fault log. This function is identical to the clear button in the Fault log in the RTCU M2M Studio.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	

<b>rmonFaultGetText()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFaultGetText(unsigned char fault, char* FaultText, int bufsize)	
<b>Description</b>	This function retrieves the text message for a Fault code	
<b>Input</b>	<b>fault</b>	The fault code
	<b>bufsize</b>	The size of the buffer where the text is to be stored. If bufsize exceeds the number of characters in the fault text, only the number of characters present will be put in the output buffer.
<b>Output</b>	<b>FaultText</b>	0 (zero) terminated string containing the fault message
<b>Returns</b>	rmonOK	

<b>rmonSoftwareUpgrade()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonSoftwareUpgrade(HRMONCON hCon, char string[35], int *res)	
<b>Description</b>	Upgrades the RTCU device.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>String</b>	0 (zero) terminated string. The upgrade key.
<b>Output</b>	<b>res</b>	The type of upgrade performed.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonError, rmonIllegalTarget	
	<b>Value</b>	<b>Description</b>
	0	Not upgraded / Wrong upgrade key
	1	GPRS enabled
	2	Device is programmable
	3	LCD display enabled
	4	Clear password.
	5	Battery enabled.
	6	FMI support enabled.
	7 - 9	Not used
	10	Citect SCADA enabled
	11	Web enabled

<b>rmonFlexOption()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonFlexOption(HRMONCON hCon, char string[12])	
<b>Description</b>	This function will enable certain options in the device.	
<b>Input</b>	<b>hCon</b>	Handle to connection
	<b>string</b>	Zero terminated string. The option key.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonError	

<b>rmonStatisticsRead()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State
<b>Synopsis</b>	rmonRet __stdcall rmonStatisticsRead(HRMONCON hCon, rmonUnitStatistics *data)	
<b>Description</b>	Fetches the device statistics from the RTCU. The size parameter in the structure must be set to the size of the structure (sizeof) before this function is called.	
<b>Input</b>	<b>hCon</b>	Handle to connection
<b>Output</b>	<b>data</b>	The function fills this structure with the statistics.
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError	
	<pre> typedef struct {     short          size;     short          max_temp;     short          avg_temp;     short          min_temp;     unsigned long  num_boot;     unsigned long  time_run;     unsigned long  time_bat;     unsigned long  time_chg;     unsigned long  gw_connect;     unsigned long  gw_incoming;     unsigned long  gw_outgoing;     unsigned long  gw_alive;     unsigned long  gprs_connect;     unsigned long  gprs_send;     unsigned long  gprs_receive;     unsigned long  gps_nofix;     unsigned long  gps_2dfix;     unsigned long  gps_3dfix; } rmonUnitStatistics; </pre>	
	The structure variables hold the following statistics:	
	<b>size</b>	<b>The size of the structure.</b>
	<b>max_temp</b>	The highest temperature measured in the device, represented in 0.01 deg. Celsius.
	<b>avg_temp</b>	The average temperature measured in the device, represented in 0.01 deg. Celsius.
	<b>min_temp</b>	The lowest temperature measured in the device, represented in 0.01 deg. Celsius.
	<b>num_boot</b>	The number of times the device has rebooted.
	<b>time_run</b>	The number of minutes the device has been operating.
	<b>time_bat</b>	The number of minutes the device has been operating from the backup battery.
	<b>time_chg</b>	The number of minutes the device have been charging the backup battery.
	<b>gw_connect</b>	The number of times the device has tried to connect to RCH.
	<b>gw_incoming</b>	The number of incoming transactions from the RCH.
	<b>gw_outgoing</b>	The number of outgoing transactions to the RCH.
	<b>gw_alive</b>	The number of keep-alive transactions the device has sent to the RCH.
	<b>gprs_connect</b>	The number of times the device successfully connected to GPRS.
	<b>gprs_send</b>	The number of bytes send over the GPRS connection.
	<b>gprs_receive</b>	The number of bytes received from the GPRS connection.
	<b>gps_nofix</b>	The number of times the GPS module has reported 'No fix'.

<b>gps_2dfix</b>	The number of times the GPS module has reported '2D fix'.
<b>gps_3dfix</b>	The number of times the GPS module has reported '3D fix'.

<b>rmonGetUnitState()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State												
<b>Synopsis</b>	rmonRet __stdcall rmonGetUnitState(HRMONCON hCon, rmoncbunitstate pfunc, void* uptr)													
<b>Description</b>	Read the execution state from the RTCU device and return it in a call-back function													
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> <tr> <td><b>pfunc</b></td> <td>Function that is called with the name and description of a serial port.</td> </tr> <tr> <td><b>uptr</b></td> <td>A user defined argument that is included in the callback function.</td> </tr> </table>		<b>hCon</b>	Handle to connection	<b>pfunc</b>	Function that is called with the name and description of a serial port.	<b>uptr</b>	A user defined argument that is included in the callback function.						
<b>hCon</b>	Handle to connection													
<b>pfunc</b>	Function that is called with the name and description of a serial port.													
<b>uptr</b>	A user defined argument that is included in the callback function.													
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonError The call-back function is defined as follows:													
	<pre>typedef void (__stdcall *rmoncbunitstate)(void* uptr, int state); state:</pre> <table border="1"> <thead> <tr> <th>Symbolic name</th> <th>Value</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td><b>RMON_STATE_RUN</b></td> <td>1</td> <td>RTCU is running</td> </tr> <tr> <td><b>RMON_STATE_HALT</b></td> <td>2</td> <td>RTCU is halted</td> </tr> <tr> <td><b>RMON_STATE_FAULT</b></td> <td>3</td> <td>RTCU is faulted</td> </tr> </tbody> </table>		Symbolic name	Value	Description	<b>RMON_STATE_RUN</b>	1	RTCU is running	<b>RMON_STATE_HALT</b>	2	RTCU is halted	<b>RMON_STATE_FAULT</b>	3	RTCU is faulted
Symbolic name	Value	Description												
<b>RMON_STATE_RUN</b>	1	RTCU is running												
<b>RMON_STATE_HALT</b>	2	RTCU is halted												
<b>RMON_STATE_FAULT</b>	3	RTCU is faulted												

<b>rmonGetPowerInformation()</b>		<b>RTCU architecture:</b> All <b>Called in:</b> Connected State		
<b>Synopsis</b>	rmonRet __stdcall rmonGetPowerInformation(HRMONCON hCon, rmon_power_info_t *data)			
<b>Description</b>	Read the current environment from RTCU device.			
<b>Input</b>	<table border="1"> <tr> <td><b>hCon</b></td> <td>Handle to connection</td> </tr> </table>		<b>hCon</b>	Handle to connection
<b>hCon</b>	Handle to connection			
<b>Output</b>	<table border="1"> <tr> <td><b>data</b></td> <td>The function fills this structure with the environment data</td> </tr> </table>		<b>data</b>	The function fills this structure with the environment data
<b>data</b>	The function fills this structure with the environment data			
<b>Returns</b>	rmonOK, rmonComError, rmonIllegalHandle, rmonError			
	<pre>typedef struct {     short          sof;     short          temp;     unsigned short sup_volt;     unsigned char  sup_type;     unsigned char  batt_type;     unsigned char  batt_level;     unsigned char  batt_charging;     unsigned char  chg_enable;     unsigned char  run_on_batt;     char           ext_disable;     char           s0_enable;     char           dcout1_enable;     char           dcout2_enable;     char           lcd_power; } rmon_power_info_t;</pre>			

The structure variables hold the following information:

<b>sof</b>	<b>The size of the structure. The size must be set before calling the function.</b>
<b>temp</b>	The temperature in the device. (scaled by 100)
<b>sup_volt</b>	The power supply voltage. (scaled by 10)
<b>sup_type</b>	The power supply type.
<b>batt_type</b>	The type of battery present.
<b>batt_level</b>	The battery power level. (0..5)
<b>batt_charging</b>	The battery is being charged.
<b>chg_enable</b>	The battery charger is enabled.
<b>run_on_batt</b>	The device will continue to run from the battery if the supply is removed.
<b>ext_disable</b>	The external power is disabled.
<b>s0_enable</b>	S0 mode is enabled for digital inputs.
<b>dcout1_enable</b>	The DC-OUT 1 is enabled.
<b>cdout2_enable</b>	The DC-OUT 2 is enabled.
<b>LCD_power</b>	The display is powered.

Supply type:

Symbolic name	Value	Description
<b>RMON_SUPPLY_BAT</b>	1	Operating on internal battery.
<b>RMON_SUPPLY_DC</b>	2	Operating on external DC power.
<b>RMON_SUPPLY_AC</b>	3	Operating on external AC power.

Battery type:

Symbolic name	Value	Description
<b>RMON_BAT_NONE</b>	0	No battery.
<b>RMON_BAT_NONCHG</b>	1	Non-chargeable battery
<b>RMON_BAT_LOW</b>	2	Low capacity battery
<b>RMON_BAT_HIGH</b>	3	High capacity battery
<b>RMON_BAT_SUPER</b>	4	Super capacity battery

The enable or disable type:

Value	Description
<b>-1</b>	Not available
<b>0</b>	Disabled
<b>1</b>	Enabled

## Appendix A, simple application

```

//-----
// Small RTCUCSP.DLL sample program
//-----
#include <windows.h>
#include <process.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <rtcucsp.h>

//-----
// Connection handle
//-----
HRMONCON hCon;

//-----
// Receive any incoming Debug messages from RTCU
//-----
static void thDebug(void *arg) {
    char buffer[512];
    int rc;
    for (;;) {
        // Wait for any debug messages from device
        rc=rmonReceiveDebugMsg(hCon, buffer, sizeof(buffer));
        if (rc==0) {
            // Just print the debug message
            printf("Debug::[%s]\n", buffer);
        } else {
            Sleep(50);
        }
    }
}

//-----
// Callback function that will be called by rmonFirmwareUpload()
// to report progress in the upload process
//-----
static int RMONCC cbfuncFW(void* uptr,int percentage) {
    printf("Firmware upload finished: %3i\r", percentage);
    return 0;
}

//-----
// Callback function that will be called by rmonApplicationUpload()
// to report progress in the upload process
//-----
static int RMONCC cbfuncApp(void* uptr,int percentage) {
    printf("Application upload finished: %3i\r", percentage);
    return 0;
}

//-----
// Callback function that will be called by rmonVoiceUpload()
// to report progress in the upload process
//-----
static int RMONCC cbfuncVoice(void* uptr,int percentage) {
    printf("Voice upload finished: %3i\r", percentage);
    return 0;
}

//-----
// the main program
//-----
int main(int argc,char** argv) {
    int rc;

    puts("The RTCU Demo project is running\n\r");
}

```

## RTCUCS Communication Support Package, Version 4.60

```

// Open RTCUCSP library
rmonOpen();

// Open a connection
hCon = rmonOpenConnection();

// Select which comports to use for local and remote connections. (Use COM0 to not use the port)
rmonSetComport(hCon, "USB1", "COM0");

// Connect to device via cable
rmonConnect(hCon, "");

// Start the listener thread for incoming Debug messages
_beginthread(thDebug, 0, NULL);

// Wait for a connection to a RTCU device
while (1) {
    // Check and wait for connection (can be RMONCON_LOCAL, RMONCON_REMOTE, RMONCON_GW or
    RMONCON_NONE)
    if (rmonConnected(hCon) != RMONCON_NONE)
        break;
    Sleep(300);
}
printf("Connected to device.\n");

// Try to authenticate with an empty password:
rc=rmonAuthenticate(hCon, "");
switch (rc) {
    case rmonDenied: printf("Authenticate with empty password denied. Use correct password !\n");
break;
    case rmonOK: printf("Authenticate with empty password accepted !\n"); break;
}

if (rc==rmonDenied) {
    printf("Not able to logon to device !\n");
    return 1;
}

// Get information about the device
int targetid, firmwareversion;
rmonGetTargetInfo(hCon, &targetid, &firmwareversion);
printf("Target ID=%i, Firmware version=0x%X\n", targetid, firmwareversion);

// Get the serial number of the device
unsigned long SerialNumber;
rmonGetSerialNumber(hCon, &SerialNumber);
printf("Serialnumber of device is %09i\n", SerialNumber);

// Use this to upload new firmware to the device:
//printf("\nrc=%i\n", rmonFirmwareUpload(hCon, "D:\\FirmwareFile.bin", cbfuncFW, (void*)0));

// Use the following to upload a new application and voice messages:
/*
rmonHalt(hCon);
printf("\nrc=%i\n", rmonApplicationUpload(hCon, "D:\\APP\\APP.VSX", cbfuncApp, NULL));
printf("\nrc=%i\n", rmonVoiceUpload(hCon, "C:\\APP\\APP.PRJ", cbfuncVoice, NULL));
rmonReset(hCon);
*/

printf("\n\nPress any key to end program...\n\n");
for (;;) {
    if (_getch()) break;
}
// Close connection after use
rmonCloseConnection(hCon);

return 0;
}

```

## Appendix B, RTCUPROG application

The RTCUPROG program is a complete Microsoft Visual Studio C++ 2019 project. This program demonstrates all aspects in making a robust application that will manage the connection to both local and remote RTCU device. The application allows the user to upload new firmware to a device, or upload a complete new project to the RTCU device.