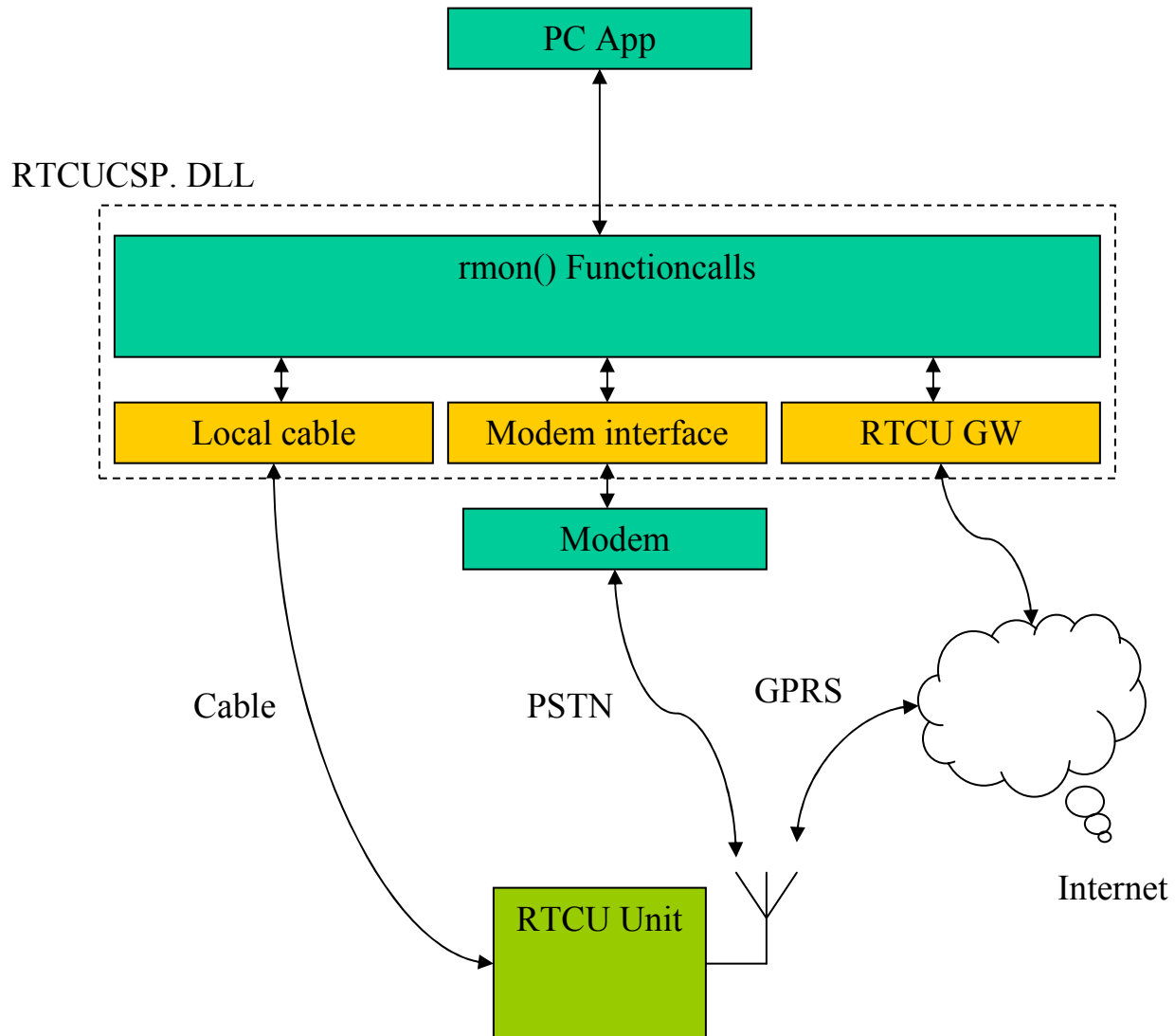


# RTCUCSP Communication Support Package

Version 1.38



## Table of Contents

<b>Introduction</b>	<b>6</b>
Graphic illustration of the library	7
<b>Differences from RTCU Communication Deployment Package</b>	<b>8</b>
<b>Contents of package</b>	<b>9</b>
<b>Interface and state diagram</b>	<b>10</b>
Initializing the library	10
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.	11
<b>Functions in the RTCUCSP.DLL library</b>	<b>13</b>
Return codes	13
<b>Initialization/Configuration</b>	<b>14</b>
<b>rmonOpen()</b>	<b>14</b>
<b>rmonClose()</b>	<b>14</b>
<b>rmonGetVer()</b>	<b>14</b>
<b>rmonSetMaxConnections()</b>	<b>15</b>
<b>rmonSetGWParameters()</b>	<b>15</b>
<b>rmonSetGWParametersAdv()</b>	<b>16</b>
<b>rmonGetPortList()</b>	<b>16</b>
<b>rmonEnumeratePorts()</b>	<b>17</b>
<b>rmonOpenConnection()</b>	<b>17</b>
<b>rmonCloseConnection()</b>	<b>17</b>
<b>rmonSetComport()</b>	<b>18</b>
<b>rmonSetModemInit()</b>	<b>18</b>
<b>rmonSetRemoteBaudrate()</b>	<b>19</b>
<b>rmonConnect()</b>	<b>19</b>
<b>rmonDisconnect()</b>	<b>20</b>
<b>rmonConnected()</b>	<b>20</b>
<b>rmonAuthenticate()</b>	<b>21</b>
<b>rmonGetComStatistics()</b>	<b>22</b>
<b>rmonResetComStatistics()</b>	<b>22</b>
<b>Program/Firmware upload</b>	<b>23</b>
<b>rmonFirmwareUpload()</b>	<b>23</b>

<b>rmonFirmwareStartUpload()</b>	24
<b>rmonFirmwareResumeUpload()</b>	25
<b>rmonApplicationUpload()</b>	26
<b>rmonApplicationStartUpload()</b>	27
<b>rmonApplicationResumeUpload()</b>	28
<b>rmonVoiceUpload()</b>	29
<b>rmonNumOfVoiceMessages()</b>	29
<b>Manipulation of Persistent memory</b>	30
<b>rmonReadPersistentFRAM()</b>	30
<b>rmonWritePersistentFRAM()</b>	31
<b>rmonReadPersistentFLASH()</b>	32
<b>rmonWritePersistentFLASH()</b>	33
<b>rmonGetXFLASHSize()</b>	33
<b>rmonReadPersistentXFLASH()</b>	34
<b>rmonWritePersistentXFLASH()</b>	35
<b>Datalogger</b>	36
<b>rmonLogFirst()</b>	36
<b>rmonLogLast()</b>	36
<b>rmonLogReadExt()</b>	37
<b>rmonLogGetValuesPerRecord()</b>	38
<b>rmonLogClear()</b>	38
<b>rmonLogGotoLinsec()</b>	39
<b>rmonLogReadByTag()</b>	40
<b>rmonLogSeek()</b>	41
<b>I/O system functions</b>	42
<b>rmonReadIOMemory()</b>	42
<b>rmonWriteIOMemory()</b>	43
<b>rmonGetIOState()</b>	44
<b>rmonSetIOState()</b>	45
<b>rmonGetIOCount()</b>	46
<b>Real time clock</b>	47
<b>rmonGetRTC()</b>	47
<b>rmonSetRTC()</b>	48
<b>GSM/SMS functions</b>	49
<b>rmonGetIMEI(), rmonGetIMSI(), rmonGetICCID()</b>	49
<b>rmonSendSMS()</b>	50
<b>rmonReceiveSMS()</b>	51
<b>rmonReceiveSMSEnable()</b>	51
<b>rmonGetGSMSignalLevel()</b>	52
<b>rmonSetAllowedCallerList()</b>	52
<b>rmonGetAllowedCallerList()</b>	53

<b>rmonSetGSMPIN()</b>	<b>54</b>
<b>rmonGetGSMPIN()</b>	<b>55</b>
<b>Filesystem functions</b>	<b>56</b>
<b>rmonMediaPresent()</b>	<b>57</b>
<b>rmonMediaWriteprotected()</b>	<b>57</b>
<b>rmonMediaOpen()</b>	<b>58</b>
<b>rmonMediaClose()</b>	<b>58</b>
<b>rmonMediaQuickformat()</b>	<b>59</b>
<b>rmonMediaEject()</b>	<b>59</b>
<b>rmonMediaInformation()</b>	<b>60</b>
<b>rmonFSStatusLED()</b>	<b>61</b>
<b>rmonDirCreate()</b>	<b>61</b>
<b>rmonDirChange()</b>	<b>62</b>
<b>rmonDirCurrent()</b>	<b>62</b>
<b>rmonDirCatalog()</b>	<b>63</b>
<b>rmonDirDelete()</b>	<b>63</b>
<b>rmonFileCreate()</b>	<b>64</b>
<b>rmonFileOpen()</b>	<b>64</b>
<b>rmonFileExists()</b>	<b>65</b>
<b>rmonFileRename()</b>	<b>65</b>
<b>rmonFileDelete()</b>	<b>66</b>
<b>rmonFileStatus()</b>	<b>66</b>
<b>rmonFileGetInfo()</b>	<b>67</b>
<b>rmonFileSeek()</b>	<b>67</b>
<b>rmonFilePosition()</b>	<b>68</b>
<b>rmonFileRead()</b>	<b>68</b>
<b>rmonFileReadString()</b>	<b>69</b>
<b>rmonFileWrite()</b>	<b>69</b>
<b>rmonFileWriteString()</b>	<b>70</b>
<b>rmonFileWriteStringNL()</b>	<b>70</b>
<b>rmonFileClose()</b>	<b>71</b>
<b>rmonFileFlush()</b>	<b>71</b>
<b>Misc. functions</b>	<b>72</b>
<b>rmonReset()</b>	<b>72</b>
<b>rmonHalt()</b>	<b>72</b>
<b>rmonGetSerialNumber()</b>	<b>73</b>
<b>rmonIsUnitProgrammable()</b>	<b>73</b>
<b>rmonVer()</b>	<b>74</b>
<b>rmonSetPassword()</b>	<b>74</b>
<b>rmonGetTargetInfo()</b>	<b>75</b>
<b>rmonGetTargetProfile()</b>	<b>76</b>
<b>rmonReceiveDebugMsg()</b>	<b>77</b>
<b>rmonGetDebugEnable()</b>	<b>77</b>

<b>rmonSetDebugEnable()</b>	78
<b>rmonVoiceMessagesAbove64K()</b>	78
<b>rmonGetAppInfo()</b>	79
<b>rmonGetGPRSSettings()</b>	80
<b>rmonSetGPRSSettings()</b>	81
<b>rmonGetGatewaySettings()</b>	82
<b>rmonSetGatewaySettings()</b>	83
<b>rmonFaultLogRead()</b>	84
<b>rmonFaultLogClear()</b>	85
<b>rmonFaultLogGetText()</b>	85
<b>rmonSoftwareUpgrade()</b>	86
<b>rmonFlexOption()</b>	87
<i>Appendix A, simple application</i>	88
<i>Appendix B, RTCUPROG application</i>	91

## Introduction

This document discusses the library RTCUCSP.DLL, which enables you to establish communication with the Logic IO RTCU products. The library is a collection of functions, which will enable your own application to “talk with” the RTCU units, transfer new applications to them, upload new firmware, manipulate persistent memory, and in general perform a lot of operations, which are also accessible from within the RTCU-IDE Integrated Development Environment. The library allows you to connect to an RTCU unit thru 3 different means, the simplest being a direct cable connection. When remote connection to an RTCU unit is needed, there are two possible solutions, either thru a normal telephone modem (data call), or using TCP/IP connection via the RTCU Gateway to the RTCU unit.

Please note that if you are interested in more information about the RTCU Gateway, you should download the “GPRS Gateway Deployment Package” which is also available at the Logic IO website.

At the end of this document, you will find appendix A and appendix B. Appendix A will show you how to make a very simple basic application, mainly it shows you how to connect to a RTCU unit, and perform some simple operations. Appendix B is a complete application; this is the RTCU-PROG application. The RTCU-PROG application allows you to upload new RTCU projects and new firmware files to a RTCU unit, either thru a cable connection, thru a modem, or thru the RTCU Gateway. The application shows all the different aspects of establishing and maintaining an application with an RTCU unit, authentication and so on. Both applications will give a good hands-on experience to the library, and can also function as a good starting point for you own applications.



## Differences from RTCU Communication Deployment Package

The RTCU Communication Support Package (RTCUCSP) is the successor of the RTCU Communication Deployment Package. The primary motivation for releasing this new package is the added support for multi-session.

Multi-session support lets users connect with multiple RTCU units simultaneous via a combination of Cable, CSD (data call) or RTCU Gateway. This effectively means that for example on one session an upgrade over GPRS of a unit can occur and at the same time on another session a datalog can be retrieved over a CSD connection.

The RTCUCSP maintains the same API as the RTCU Communication Deployment Package, but to support the concept of multi-session a new session-handle parameter has been added to almost all functions.

Also the RTCUCSP does not operate with a default connection via. Cable, and such a connection must be initiated using `rmonConnect()`.

There has also been a general cleanup of function interfaces. Fewer structures are used, as similar ones have been merged, and unused parameters have been removed.

Finally, full support for Encrypted communication (with a user defined key) using the RTCU Gateway Professional has been added.

Migration of an application to use the RTCUCSP will be quite easy when the above listed differences are taking into account.

## Contents of package

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

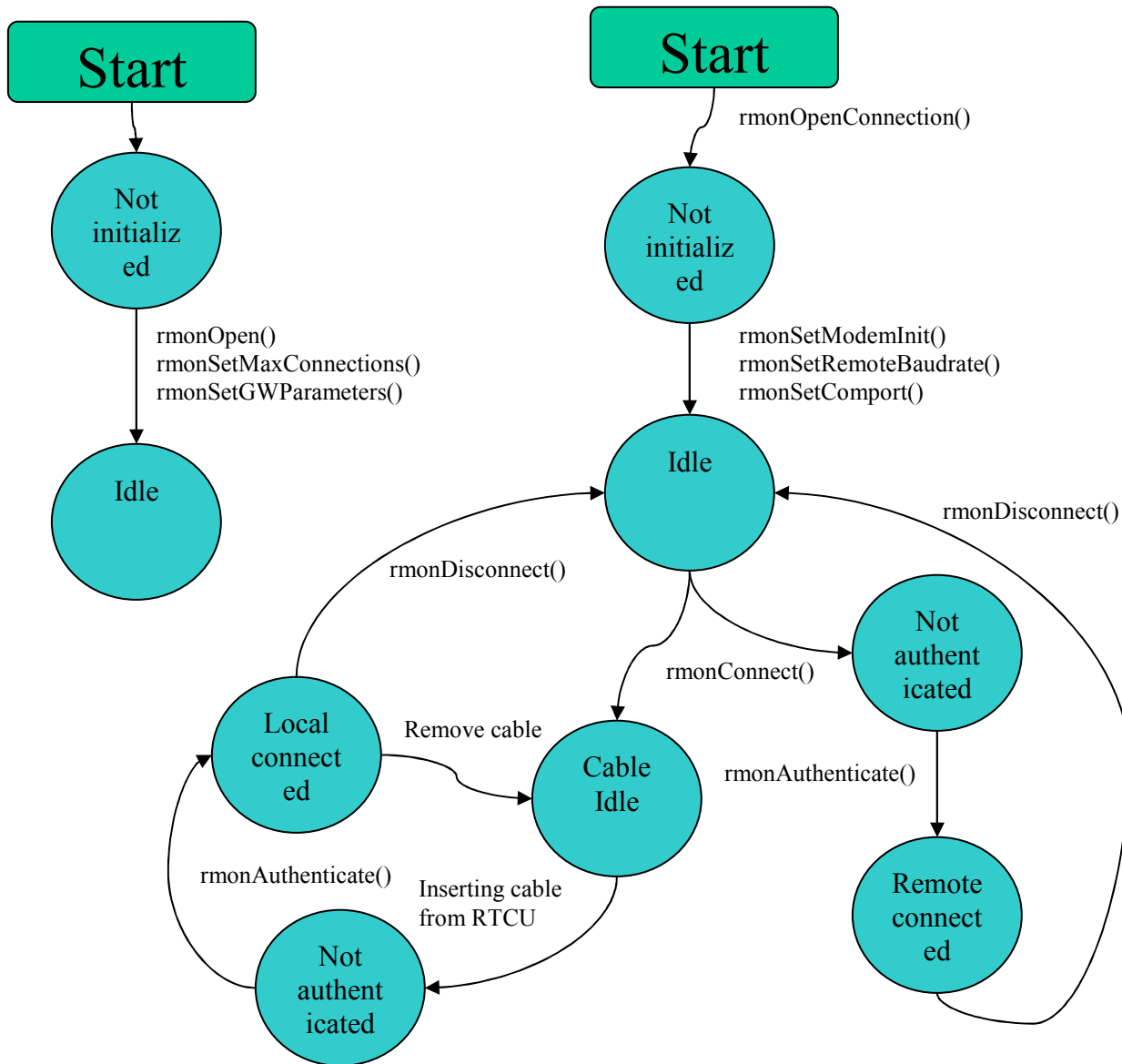
“\RTCU Communication Support Package. pdf”  
“\RTCUPROG”  
“\Library\”

This document  
The RTCUPROG application described in appendix B  
The .H, .LIB and .DLL files needed for the RTCUCSP  
library. (Please note that the RTCUCSP.DLL also uses  
other DLL's, these are included in the “Library” folder  
also)

In order to use this library, you will need to use Microsoft Visual Studio C++ 2005. For an example of which Settings etc are needed, please have a look at the RTCUPROG application included in this package.

## 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**. In order 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 `rmonSetMazConnections`
- **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 units.

## Initializing the connection.

After calling the `rmonOpenConnection`, you are in the **Not Initialized State**. In order 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 a RTCU Gateway 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 a RTCU unit either thru

- A local cable connection or
- A remote connection – (modem (CSD) or RTCU Gateway 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 unit 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 unit using the functions described.

## Closing or changing the connection.

### A. The Locally Connected State:

As you can see at the state diagram above you have **three possibilities** 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 at the state diagram above you have **three possibilities** 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 X32, large and small RTCU units, as some of the functions are not supported on the small models. The X32 models currently includes the MX2 series, DX4 series and AX9 series, the large models currently includes the M11 Series, M10 Series and A9i, and small models are D4, A5, A6, DIN, SA and M7. 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 unit 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 Gateway
rmonNotGateway	7	The function can not be used over the Gateway
rmonDenied	8	Access to unit 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	Gateway 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 RTCU unit.
rmonInterrupted	22	Background transfer was interrupted.
rmonCancelled	23	Data transfer has been cancelled.
rmonNotProgrammable	24	Attempt to program a Micro unit 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 unit.
rmonImageTooLarge	28	There are no room for the image in the RTCU unit.
rmonNotModem	29	The function can not be used over Modem connection.

## Initialization/Configuration

Functions that are usually needed before any real communication can be started with the RTCU unit 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).

---

### rmonOpen()

<b>RTCU model:</b> n/a <b>Called in:</b> Not Initialised
---

**Synopsis**                      rmonRet \_\_stdcall rmonOpen(void)

**Description**                Opens and initializes the communication system. The system can be closed again by calling rmonClose().

**Returns**                      rmonOK, rmonError, rmonInit

---

### rmonClose()

<b>RTCU model:</b> n/a <b>Called in:</b> Idle State
--

**Synopsis**                      rmonRet \_\_stdcall rmonClose(void)

**Description**                Closes communication system. The communication system must be opened with rmonOpen() in order for it to be used again.

**Returns**                      rmonOK, rmonNotInit

---

### rmonGetVer()

<b>RTCU model:</b> n/a <b>Called in:</b> Not Initialised and Idle State
--

**Synopsis**                      int \_\_stdcall rmonGetVer(void)

**Description**                Retrieve the version number of the RTCUCSP library.

**Returns**                      Library version scaled by 100.

## rmonSetMaxConnections()

**RTCU model:** n/a  
**Called in:** Not Initialised

**Synopsis** rmonRet \_\_stdcall rmonSetMaxConnections (int max\_sessions)

**Description** Set maximum number of simultaneous connections possible.  
 If this function is not called the default number of simultaneous connections will be used.

**Input**

max_sessions	The maximum number of connections. (Default: 100)
--------------	---

**Returns** rmonOK, rmonInit

## rmonSetGWParameters()

**RTCU model:** Large & X32  
**Called in:** Not Initialised

**Synopsis** rmonRet \_\_stdcall rmonSetGWParameters(const unsigned short Port, const unsigned long MyNodeID, const char\* IP, const char\* Key)

**Description** If connection to a remote RTCU is to be done using the RTCU Gateway, some parameters have to be set before this is possible. These parameters are set using this function. Please see the online help to the RTCU-IDE Integrated Development Environment for a description of these parameters (Menu: Unit -> Communication -> Connect via RTCU Gateway).

**Input**

Port	Gateway port (please refer to rmonConnect() for making a connection thru the RTCU Gateway)
MyNodeID	Node ID (can be set to 0, this will allow the Gateway to issue a "dynamic" ID to this node).
IP	IP address of the RTCU Gateway
Key	Key value (must be set to the same value as set in the RTCU Gateway)

**Returns** rmonOK, rmonInit

## rmonSetGWParametersAdv()

**RTCU model:** Large & X32  
**Called in:** Not Initialised

**Synopsis**                    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)

**Description**                This function is used to set the advanced parameters for connecting to a remote RTCU unit using the RTCU Gateway. Please see the online help to the RTCU-IDE Integrated Development Environment for a description of these parameters (Menu: Unit - > Communication -> Connect via RTCU Gateway).

### Input

CryptKey	Encryption key. To use the default key, set all 16 bytes to 0 (zero).
gw_max_connection_attempt	Maximum connection attempts. Default value: 3. Range: 1-60
gw_max_send_req_attempt	Maximum transmission attempts. Default value: 3. Range: 1-60
gw_response_timeout	Response timeout. Default value: 30. Range: 5-60
gw_alive_freq	Keep alive frequency. Default value: 60. Range: 0-60000

**Returns**                      rmonOK, rmonInit

## rmonGetPortList()

**RTCU model:** n/a  
**Called in:** Not Initialised and Idle State

**Synopsis**                      rmonRet \_\_stdcall rmonGetPortList (char name[RMON\_MAXPORTS][9], int\* size)

**Description**                Retrieve the names and number of comports.

### Output

name	Array of ASCII strings with the names of the comports.
size	The number of comports present.

**Returns**                      rmonOK, rmonNoData

## rmonEnumeratePorts()

**RTCU model:** n/a  
**Called in:** All states

**Synopsis**                    rmonRet \_\_stdcall rmonEnumeratePorts (rmoncbserialport cbFunc, void \*arg, int \*count)

**Description**                Enumerate the serial ports present.

### Input

cbFunc	Function that is called with the name and description of a serial port.
arg	A user defined argument that is included in the callback function.

### Output

count	The number of serial ports enumerated.
-------	--

**Returns**                    rmonOK

The call-back function is defined as follows:

```
typedef void (__stdcall *rmoncbserialport)(void *arg, const char *name, const char *desc);
```

## rmonOpenConnection()

**RTCU model:** Large, Small & X32  
**Called in:** Idle State

**Synopsis**                    HRMONCON \_\_stdcall rmonOpenConnection (void)

**Description**                Open a new connection session.

**Returns**                    Handle to connection or HRMONCON\_ILLEGAL if no connection.

## rmonCloseConnection()

**RTCU model:** Large, Small & X32  
**Called in:** Not Initialised and Idle State

**Synopsis**                    rmonRet \_\_stdcall rmonCloseConnection (HRMONCON hCon)

**Description**                Close a connection session.

### Input

hCon	Handle to connection
------	----------------------

**Returns**                    rmonOK, rmonIllegalHandle

## rmonSetComport()

**RTCU model:** Large, Small & X32  
**Called in:** Not Initialised and Idle State

**Synopsis**                      rmonRet \_\_stdcall rmonSetComport(HRMONCON hCon, const char\* LocalPort, const char\* RemotePort)

**Description**                      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 thru a modem. These ports are configured using this call. If one of the ports is not to be used, simply set it to "COM0".  
 To configure an USB cable connection, use the port names "USB1" thru "USB8".

**Input**

hCon	Handle to connection.
LocalPort	Name of the COM port to be used for direct cable connection.
RemotePort	Name of the COM port to be used for remote connection thru a modem.

**Returns**                              rmonOK, rmonConnection, rmonIllegalHandle

## rmonSetModemInit()

**RTCU model:** Large, Small & X32  
**Called in:** Not Initialised and Idle State

**Synopsis**                              rmonRet \_\_stdcall rmonSetModemInit(HRMONCON hCon, const char\* atcmd)

**Description**                              Specifies initialisation string to modem for usage in remote connection. A list of common initialization string for different types of modems can be seen in the RTCU-IDE Integrated Development Environment (menu: Settings -> Setup).

**Input**

hCon	Handle to the connection.
Atcmd	Initialisation string for modem

**Returns**                                      rmonOK, rmonConnection, rmonIllegalHandle

## rmonSetRemoteBaudrate()

**RTCU model:** Large, Small & X32  
**Called in:** Not Initialised and Idle State

**Synopsis** `rmonRet __stdcall rmonSetRemoteBaudrate(HRMONCON hCon, int baud)`

**Description** This function sets the baud rate that will be used when communication is established to a remote unit thru 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 unit (which can be influenced by setting the appropriate settings in the `rmpnSetModemInit()`).

**Input**

hCon	Handle to connection
Baud	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.

**Returns** `rmonOK, rmonConnection, rmonIllegalHandle`

## rmonConnect()

**RTCU model:** Large, Small & X32  
**Called in:** Idle

**Synopsis** `rmonRet __stdcall rmonConnect(HRMONCON hCon, const char* phonenumber)`

**Description** This function is used to establish a connection to a RTCU unit. The connection can be either over cable, thru a modem, or thru the RTCU Gateway. If the remote RTCU is to be contacted thru a modem, simply use the telephone number of the SIM card in the RTCU, if connection is thru the RTCU Gateway, the units serial number (nodeid) is to be used, prefixed with a "@" character! To establish a connection over cable leave the phonenumber empty.

**Input**

hCon	Handle to connection
phonenumber	The phone number of the SIM card in the remote RTCU if connection is thru modem, or the serial number of the RTCU (prefixed with "@") if connection is thru RTCU Gateway.

**Returns** `rmonOK, rmonIllegalHandle, rmonConnection, rmonDenied, rmonComError, rmonError`

## rmonDisconnect()

**RTCU model:** Large, Small & X32  
**Called in:** Remotely Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonDisconnect(HRMONCON hCon)

**Description**                 Disconnect a connection to a RTCU unit.

**Input**

hCon	Handle to connection
------	----------------------

**Returns**                      rmonOK, rmonIllegalHandle, rmonError

## rmonConnected()

**RTCU model:** Large, Small & X32  
**Called in:** All states except Not Initialised State

**Synopsis**                      rmonRet \_\_stdcall rmonConnected(HRMONCON hCon)

**Description**                 rmonConnected() returns the type of connection that is currently (if any) established with the RTCU unit.

**Input**

hCon	Handle to connection
------	----------------------

**Returns**                      Type of connection, see below

Type of connection:

Symbolic name	Value	Description
RMONCON_NONE	0	Currently not connected
RMONCON_LOCAL	1	Connected using cable
RMONCON_REMOTE	2	Connected thru a modem
RMONCON_GW	3	Connected thru the RTCU Gateway

## rmonAuthenticate()

**RTCU model:** Large, Small & X32  
**Called in:** Not Authenticated State

**Synopsis**                      rmonRet \_\_stdcall rmonAuthenticate (HRMONCON hCon, const char password[21])

**Description**                      if there is established a (new) connection with a RTCU unit, either local or remote, the first thing to do, is for the application to authenticate itself for the RTCU unit. This is done using this function, and must be done, BEFORE any other communication with the unit can take place.  
If there is no password set in the RTCU unit, this function must still be called, just with an empty string "" as the password.

### Input

hCon	Handle to connection
password	Password, zero terminated ASCII string

**Returns**                              rmonOK, rmonComError, rmonIllegalHandle, rmonIllegalTarget, rmonTargetError, rmonDenied

## rmonGetComStatistics()

**RTCU model:** Large, Small & X32  
**Called in:** All states

**Synopsis**                    rmonRet \_\_stdcall rmonGetComStatistics(HRMONCON hCon, struct rmonComStatistics\* pStat).

**Description**                rmonGetComStatistics() returns the statistics of the Cable and Modem connection types.

### Input

hCon	Handle to connection
------	----------------------

### Output

pStat	Structure containing the statistics for the connection
-------	--

**Returns**                      rmonOK, rmonIllegalHandle, rmonConnection, rmonNotGateway, rmonError

```
struct rmonComStatistics {
    int RXBytes;           // Number of bytes sent
    int TXBytes;           // Number of bytes received.
    int RXPackets;        // Number of packets received.
    int TXPackets;        // Number of packets sent.
    int RXNAK;            // Number of frames NAK'ed by transmitter.
    int TXNAK;            // Number of frames NAK'ed by receiver.
    int RXTimeout;        // Number of receive timeouts.
    int TXTimeout;        // Number of TX packets with timeout.
    int RXCRC;            // Number of packets with CRC error.
};
```

## rmonResetComStatistics()

**RTCU model:** Large, Small & X32  
**Called in:** All states

**Synopsis**                    rmonRet \_\_stdcall rmonResetComStatistics(HRMONCON hCon)

**Description**                This function resets the communication statistics for Cable and Modem connections.

### Input

hCon	Handle to connection
------	----------------------

**Returns**                      rmonOK, rmonIllegalHandle, rmonConnection, rmonNotGateway

## Program/Firmware upload

The following functions are used for uploading new applications, voice messages and firmware to a RTCU unit: All functions reports 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.

### rmonFirmwareUpload()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                    rmonRet \_\_stdcall rmonFirmwareUpload(HRMONCON hCon, char \*FirmwareFilename, rmoncbprogress cbfunc, void \*uptr);

**Description**                Upload firmware to RTCU.  
 Function takes a .BIN firmware file, and transfers it to a RTCU unit. The function will halt execution in the unit, upload the new firmware file, and after the transfer, it will reset the unit. Note that to upload a new firmware to a RTCU unit when the connection to it is via the RTCU Gateway, you need to use background update (see rmonFirmwareStartUpload).

#### Input

hCon	Handle to connection
FirmwareFilename	Firmware file
cbfunc	Call-back function for progress
uptr	User data that will be passed to call-back function when called

**Returns**                    rmonOK, rmonComError, rmonIllegalHandle, rmonIllegalTarget, rmonNotGateway, rmonNoMonitorMode, rmonErrorReset, rmonCancelled, rmonFileNotFound, rmonIllegalFile, rmonOldFormat, rmonNotModem

## rmonFirmwareStartUpload()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFirmwareStartUpload(HRMONCON hCon, const char* Filename, struct rmonBGReport* Report, rmoncbprogress cbfunc, void* uptr );`

**Description** Upload firmware to RTCU.  
 Function takes a .BIN firmware file, and transfers it to a RTCU unit. Unlike rmonFirmwareUpload the function will not halt execution in the unit, 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 unit has been reset.  
 Note that the voice memory in the unit 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.

### Input

hCon	Handle to connection
Filename	Zero terminated string with the firmware filename
Report	A structure containing progress status (see definition below)
cbfunc	Call-back function for progress
uptr	User data that will be passed to call-back function when called

**Returns** rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonCancelled, rmonNoBackground, rmonFileNotFound, rmonOldFormat, rmonIllegalFile, rmonIllegalTarget, rmonInterrupted

```
struct rmonBGReport {
    unsigned long extSeg;    // External segment written
    unsigned long intSeg;   // Internal segment written
    unsigned long headSeg;  // Header segment written
};
```

## rmonFirmwareResumeUpload()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFirmwareResumeUpload(HRMONCON hCon, const char* Filename, struct rmonBGReport* Report, rmoncbprogress cbfunc, void* uptr );`

**Description** Upload firmware to RTCU.  
 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.  
 Note that this function cannot be used to start a new upload, only complete an interrupted upload.  
 Note that the voice memory in the unit is used and voice data must be uploaded again.

### Input

hCon	Handle to connection
Filename	Zero terminated string with the firmware filename
Report	A structure containing progress status (see definition below)
cbfunc	Call-back function for progress
uptr	User data that will be passed to call-back function when called

**Returns** `rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonCancelled, rmonNoBackground, rmonFileNotFound, rmonOldFormat, rmonIllegalFile, rmonIllegalTarget, rmonInterrupted`

```
struct rmonBGReport{
    unsigned long extSeg;    // External segment written
    unsigned long intSeg;   // Internal segment written
    unsigned long headSeg;  // Header segment written
};
```

## rmonApplicationUpload()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonApplicationUpload(HRMONCON hCon, char *Filename, rmoncbprogress cbfunc, void *uptr);`

**Description** Uploads application to RTCU  
 Function takes a .VSX or a .PSX file, and transfers it to a RTCU.  
 Please note that the execution of the VPL program in the RTCU unit **must** be halted with `rmonHalt()`, **before** calling this function ! The RTCU will have to be reset after the transfer, to start the new application.

### Input

hCon	Handle to connection
Filename	Zero terminated string with the filename of the application
cbfunc	Call back function for progress
uptr	User data that will be passed to call back function when called

**Returns** `rmonOK`, `rmonComError`, `rmonIllegalHandle`, `rmonErrorHalt`, `rmonFileNotFound`, `rmonIllegalFile`, `rmonNotProgrammable`, `rmonCancelled`, `rmonTargetError`, `rmonIllegalTarget`, `rmonImageTooLarge`

## rmonApplicationStartUpload()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonApplicationStartUpload(HRMONCON hCon, const char* Filename, struct rmonBGReport* Report, rmoncbprogress cbfunc, void *uptr);`

**Description** Uploads application to RTCU  
 Function takes a Report containing a .VSX or a .PSX 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`.  
 The newly uploaded application is used after the unit has been reset.  
 Note that the voice memory in the unit 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.

### Input

hCon	Handle to connection
Filename	Zero terminated string with the filename of the application
Report	A structure containing progress status (see definition below)
cbfunc	Call back function for progress
uptr	User data that will be passed to call back function when called

**Returns** `rmonOK`, `rmonComError`, `rmonIllegalHandle`, `rmonTargetError`, `rmonIllegalTarget`, `rmonCancelled`, `rmonNoBackground`, `rmonFileNotFound`, `rmonIllegalFile`, `rmonNotProgrammable`, `rmonInterrupted`, `rmonImageTooLarge`

```
struct rmonBGReport {
    unsigned long extSeg;    // External segment written
    unsigned long intSeg;   // Internal segment written
    unsigned long headSeg;  // Header segment written
};
```

## rmonApplicationResumeUpload()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonApplicationResumeUpload(HRMONCON hCon, const char* Filename, struct rmonBGReport* Report, rmoncbprogress cbfunc, void *uptr);`

**Description** Uploads application to RTCU  
 Function takes a report containing a .VSX or a .PSX 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.  
 Note that this function cannot be used to start a new upload, only complete an interrupted upload.  
 Note that the voice memory in the unit is used and any voice data must be uploaded again.

### Input

hCon	Handle to connection
Filename	Zero terminated string with the filename of the application
Report	A structure containing progress status (see definition below)
cbfunc	Call back function for progress
uptr	User data that will be passed to call back function when called

**Returns** rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonIllegalTarget, rmonCancelled, rmonNoBackground, rmonFileNotFound, rmonIllegalFile, rmonNotProgrammable, rmonInterrupted, rmonImageTooLarge

```
struct rmonBGReport {
    unsigned long extSeg;    // External segment written
    unsigned long intSeg;   // Internal segment written
    unsigned long headSeg;  // Header segment written
};
```

## rmonVoiceUpload()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonVoiceUpload(HRMONCON hCon, char *ProjectFilename, rmoncbprogress cbfunc, void *uptr);`

**Description** Upload Voice messages to RTCU.  
 Function transfers all voice messages associated with a RTCU project. It is important that the relative directory structure for the project is maintained as when the project was built in the RTCU-IDE environment, otherwise the function will have trouble locating the voice message files. Please note that the execution of the VPL program in the RTCU unit **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.

**Input**

hCon	Handle to connection
ProjectFilename	Project filename
cbfunc	Call back function for progress
uptr	User data that will be passed to call back function when called

**Returns** `rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonCancelled, rmonFileNotFound, rmonIllegalFile, rmonMemoryConfig, rmonImageTooLarge`

## rmonNumOfVoiceMessages()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonNumOfVoiceMessages(char *ProjectFilename, int *NumFiles);`

**Description** 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 unit.

**Input**

ProjectFilename	Project filename
NumFile	Number of voice file in PRJ file

**Returns** `rmonOK, rmonFileNotFound, rmonIllegalFile`

## Manipulation of Persistent memory

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

### rmonReadPersistentFRAM()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonReadPersistentFRAM(HRMONCON hCon, int entry, char *data, int *length, int binary);`

**Description** Read FRAM based persistent entry.  
 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.  
 Data read with this function, can be stored from VPL with the functions SaveStringF() and SaveDataF()

#### Input

hCon	Handle to connection
entry	Entry number, from 1 to 20
binary	Set to 1 if expecting binary data, 0 if string expected

#### Output

data	Buffer for data (must be large enough!)
length	The number of bytes read from the entry

**Returns** rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData, rmonError

## rmonWritePersistentFRAM()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonWritePersistentFRAM(HRMONCON hCon, int entry, char *data, int length, int binary);`

**Description** Write to FRAM based persistent entry.  
 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.  
 Data stored with this function, can be read from VPL with the functions LoadStringF() and LoadDataF().

### Input

hCon	Handle to connection
entry	Entry number, from 1 to 20
data	The data to store
length	Length of data
binary	Set to 1 if storing binary data, 0 if string

**Returns** `rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonError`

## rmonReadPersistentFLASH()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonReadPersistentFLASH(HRMONCON hCon, int entry, char *data, int *length, int binary);`

**Description** Read FLASH based persistent entry  
 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.  
 Data read with this function can be stored from VPL with the functions `SaveString()` and `SaveData()`.

### Input

hCon	Handle to connection
entry	Entry number, from 1 to 192
binary	Set to 1 if expecting binary data, 0 if string expected

### Output

data	Buffer for data (must be large enough!)
length	The number of bytes read from the entry

**Returns** `rmonOK`, `rmonComError`, `rmonIllegalHandle`, `rmonTargetError`, `rmonNoData`, `rmonError`

## rmonWritePersistentFLASH()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonWritePersistentFLASH(HRMONCON hCon, int entry, char *data, int length, int binary);`

**Description** Write to FLASH based persistent entry.  
 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.  
 Data stored with this function, can be read from VPL with the functions LoadString() and LoadData().

### Input

hCon	Handle to connection
entry	Entry number, from 1 to 192
data	The data to store
length	Length of data
binary	Set to 1 if storing binary data, 0 if string

**Returns** `rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonError`

## rmonGetXFLASHSize()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonGetXFLASHSize(HRMONCON hCon, int *size);`

**Description** Get the number of entries in extended FLASH.  
 This function is identical to the VPL function GetFlashXSize().

### Input

hCon	Handle to connection
------	----------------------

### Output

size	Number of entries in extended flash.
------	--------------------------------------

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

## rmonReadPersistentXFLASH()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonReadPersistentXFLASH(HRMONCON hCon, int entry, char *data, int *length, int binary);`

**Description** 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()`.

### Input

hCon	Handle to connection
entry	Entry number, from 1 to Size (Determined with the function <code>rmonGetXFLASHSize</code> )
binary	Set to 1 if expecting binary data, 0 if string expected

### Output

data	Buffer for data (must be large enough!)
length	The number of bytes read from the entry

**Returns** `rmonOK`, `rmonComError`, `rmonIllegalHandle`, `rmonTargetError`, `rmonError`, `rmonNoData`

## rmonWritePersistentXFLASH()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonWritePersistentXFLASH(HRMONCON hCon, int entry, char *data, int length, int binary);`

**Description** Write to extended FLASH based persistent entry. 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. Data stored with this function, can be read from VPL with the functions LoadStringX() and LoadDataX().

### Input

hCon	Handle to connection
entry	Entry number, from 1 to Size (Determined with the function rmonGetXFLASHSize)
data	The data to store
length	Length of data
binary	Set to 1 if storing binary data, 0 if string

**Returns** rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError. rmonError

## Datalogger

The built-in datalogger of the RTCU unit 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 units, please refer to the online help for the RTCU-IDE Integrated Development Environment.

### rmonLogFirst()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonLogFirst(HRMONCON hCon)

**Description**                Moves the current read pointer to the first (oldest) entry in the datalogger.

**Input**

hCon	Handle to connection
------	----------------------

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

### rmonLogLast()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonLogLast(HRMONCON hCon)

**Description**                Moves the current read pointer to the last (newest) entry in the datalogger..

**Input**

hCon	Handle to connection
------	----------------------

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

## rmonLogReadExt()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonLogReadExt(HRMONCON hCon, int operation, int* values_per_rec, int* entries_in_buffer, char* buffer);`

**Description** This function reads up to 21 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.

### Input

hCon	Handle to connection
operation	RMONLOGGET_NEXT or RMONLOGGET_PREV

### Output

values_per_rec	Number of entries in tdefExtLog logvalues (array length), maximum 8
entries_in_buffer	Number of tdefExtLog records in buffer, maximum 21
buffer	Buffer containing entries in buffer records with a tdefExtLog structure, see struct definition below

**Returns** `rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData, rmonNoMoreData`

```
typedef struct {
    signed   char year;
    unsigned char month;
    unsigned char date;
    unsigned char hour;
    unsigned char minute;
    unsigned char second;
    unsigned char tag;
    int        logvalue[8];
} tdefExtLog;
```

## rmonLogGetValuesPerRecord()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
Connected State

**Synopsis** `rmonRet __stdcall rmonLogGetValuesPerRecord(HRMONCON hCon, int* numberofvalues)`

**Description** This function returns information about how many values (up to 8) are stored at each record in the datalogger (is configure via the VPL program in the RTCU unit)

### Input

hCon	Handle to connection
------	----------------------

### Output

numberofvalues	The number of values stored in each record in the datalogger.
----------------	---

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

## rmonLogClear()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
Connected State

**Synopsis** `rmonRet __stdcall rmonLogClear(HRMONCON hCon)`

**Description** Clears data in the RTCU datalogger. The current datastructure in the RTCU datalogger is maintained.

### Input

hCon	Handle to connection
------	----------------------

**Returns** `rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNotInit`

## rmonLogGotoLinsec()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonLogGotoLinsec(HRMONCON hCon, struct rmonRTCTime timestamp, unsigned char direction)`

**Description** 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.

### Input

hCon	Handle to connection
timestamp	Complete time of record to search for, Please refer to the definition of rmonRTCTime below
direction	False (0) means backwards search, True (different from 0) means forward search

**Returns** rmonOK, rmonComError, rmonIllegalHandle, rmonNoData

## rmonLogReadByTag()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonLogReadByTag(HRMONCON hCon, int operation, unsigned char tag, int* values_per_rec, int* entries_in_buffer, char* buffer);`

**Description** This function reads up to 21 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.

### Input

hCon	Handle to connection
operation	RMONLOGGET_NEXT or RMONLOGGET_PREV
tag	The tag that is used to filter datalog entries

### Output

values_per_rec	Number of entries in tdefExtLog logvalues (array length), maximum 8
entries_in_buffer	Number of tdefExtLog records in buffer, maximum 21
buffer	Buffer containing entries in buffer records with a tdefExtLog structure, see struct definition below

**Returns** `rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData, rmonNoMoreData`

```
typedef struct {
    signed   char year;
    unsigned char month;
    unsigned char date;
    unsigned char hour;
    unsigned char minute;
    unsigned char second;
    unsigned char tag;
    int        logvalue[8];
} tdefExtLog;
```

## rmonLogSeek()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonLogSeek(HRMONCON hCon, short tag, int n)`

**Description** 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.

### Input

hCon	Handle to connection
tag	The tag to search for.
n	The number of records to move. > 0 (zero): Seek forward. = 0 (zero): No effect. < 0 (zero): Seek Backwards.

**Returns** rmonOK, rmonComError, rmonIllegalHandle, rmonNoData

## I/O system functions

This group of functions allows access to the physical in- and outputs of the RTCU unit, as well as the memory I/O system. The memory I/O system is accessible thru the VPL program as normal VAR\_INPUT/VAR\_OUTPUT variables. The variables must be configured in the RTCU-IDE job configuration to 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 16 elements in a small RTCU, 1024 elements in a large RTCU and 4096 elements for X32 based RTCU.

### rmonReadIOMemory()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonReadIOMemory(HRMONCON hCon, int location, int count, int type, void *data);`

**Description** 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.

#### Input

hCon	Handle to connection
location	This is the start location to read from, 0 based.
count	Number of memory locations to read
type	1=BOOL, 2=SINT, 3=INT, 4=DINT, this is the type of data to read from each location

#### Output

data	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)
------	--

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

## rmonWriteIOMemory()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonWriteIOMemory(HRMONCON hCon, int location, int count, int type, void *data);`

**Description** 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 unit, 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 unit itself (all I/O configured variables will be updated in each scan of the VPL program in the RTCU unit).

### Input

hCon	Handle to connection
location	This is the start location to write to, 0 based.
count	Number of memory locations to write
type	1=BOOL, 2=SINT, 3=INT, 4=DINT
data	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)

**Returns** `rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonError`

## rmonGetIOState()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonGetIOState(HRMONCON hCon, int iotype, int ioindex, int* value)`

**Description** This function is used to read the state of the physical in- and output signals in the RTCU unit. The 'iotype' indicates which input/output you are reading from, and 'ioindex' indicates which input- or output number you are reading.

### Input

hCon	Handle to connection
iotype	Select the type of I/O system you want to read from, see below
ioindex	Valid index of IO to get value from. Starts with index 0

### Output

value	Input or output value
-------	-----------------------

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

iotype:

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

## rmonSetIOState()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonSetIOState(HRMONCON hCon, int iotype, int ioidex, int value)`

**Description** This function is used to set the status of the physical output signals in the RTCU unit. The 'iotype' indicated which Output you are writing to, and 'ioidex' indicates which output number you are writing to. Please note that if the output written to, is also used as a VAR\_OUTPUT variable by the VPL program in the RTCU unit, 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 unit itself (all I/O configured variables will be updated in each scan of the VPL program in the RTCU unit).

### Input

hCon	Handle to connection
iotype	Select the type of output system you want to set, see below
ioidex	This is the number of the output, 0 based.
value	Value to set

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

iotype:

Symbolic name	Value	Description
RMON_IOTYPE_DOUT	2	Digital output
RMON_IOTYPE_AOUT	4	Analog output
RMON_IOTYPE_LED	5	LED

## rmonGetIOCount()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonGetIOCount(HRMONCON hCon, struct rmonIOCount *data)`

**Description** This function is used to read the number of inputs and outputs on the unit. (Onboard and external)

### Input

hCon	Handle to connection
------	----------------------

### Output

data	A structure that contains the I/O count
------	---

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

```
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;
```

## Real time clock

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

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
};
```

### rmonGetRTC()

**RTC model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonGetRTC(HRMONCON hCon, struct rmonRTCTime \*rtc)

**Description**                Reads the real time clock on the RTCU unit. Returns the current time in a rmonRTCTime structure.

#### Input

hCon	Handle to connection
------	----------------------

#### Output

rtc	Please refer to the definition of rmonRTCTime above
-----	---

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

## rmonSetRTC()

**RTC model:** Large, Small & X32  
**Called in:** Locally & Remotely  
Connected State

**Synopsis** rmonRet\_stdcall rmonSetRTC(HRMONCON hCon, struct rmonRTCTime \*rtc)

**Description** Sets the real time clock on the RTCU unit. The time is supplied in a rmonRTCTime structure.

### Input

hCon	Handle to connection
rtc	Please refer to the definition of rmonRTCTime above

**Returns** 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 unit's interaction with the GSM module.

Also the functions allow you to send and receive "fake" SMS messages to/from the RTCU unit. If the VPL program in the unit 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 available thru 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 unit.

`rmonGetIMEI()`,  
`rmonGetIMSI()`,  
`rmonGetICCID()`

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

### Synopsis

```
rmonRet __stdcall rmonGetIMEI(HRMONCON hCon, char *IMEInumber, int
bufsize)
rmonRet __stdcall rmonGetIMSI(HRMONCON hCon, char *IMSInumber, int
bufsize)
rmonRet __stdcall rmonGetICCID(HRMONCON hCon, char *ICCIDnumber, int
bufsize)
```

### Description

`rmonGetIMEI()`, `rmonGetIMSI()` & `rmonGetICCID()` is used for fetching the IMEI number of the GSM module, or the IMSI and ICC numbers of the SIM card installed in the RTCU unit.

### Input

hCon	Handle to connection
bufsize	Number of characters to read. If bufsize exceeds the number of characters in the IMEI, IMSI or ICC only the number of characters present will be put in the output buffer.

### Output

IMEInumber, IMSInumber or ICCIDnumber	This is where the information will be stored
---------------------------------------	--

### Returns

`rmonOK`, `rmonComError`, `rmonIllegalHandle`, `rmonTargetError`

## rmonSendSMS()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonSendSMS(HRMONCON hCon, int smstype, int messageLength, const char* message)`

**Description** The PC application can send “fake” SMS messages to the RTCU unit using this function. The RTCU will receive SMS messages with the `gsmIncomingSMS()` / `gsmIncomingPDU()`, and when the message is sent to the RTCU using this function, the `.phonenumber` variable in the `gsmIncomingSMS()` / `gsmIncomingPDU()` will indicate “9999” as the originator of the message.

### Input

hCon	Handle to connection
smstype	Type of SMS message received, see below
messageLength	Only used when smstype is RMONSMS_BINARY
message	Zero terminated AZCII string when smstype is RMONSMS_TEXT. If smstype is RMONSMS_BINARY messageLength specifies length of data in message.

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

smstype:

Symbolic name	Value	Description
RMONSMS_TEXT	0	Text based SMS message
RMONSMS_BINARY	1	Binary SMS message

## rmonReceiveSMS()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonReceiveSMS(HRMONCON hCon, int* smstype, int* dataLength, char* data)`

**Description** When the connected RTCU unit sends a SMS message to phonenumber "9999" using the either `gsmSendSMS()` or `gsmSendPDU()`, this function will receive these messages. Please note that this function is blocking, it will first return when a message is received.

**Input**

gCon	Handle to connection
------	----------------------

**Output**

smstype	Type of SMS message received, see below
dataLength	Only used when smstype = RMONSMS_BINARY
data	Zero terminated ASCII string when smstype = RMONSMS_TEXT

**Returns** `rmonOK, rmonIllegalHandle, rmonTargetError, rmonNoData`

smstype:

Symbolic name	Value	Description
RMONSMS_TEXT	0	Text based SMS message
RMONSMS_BINARY	1	Binary SMS message

## rmonReceiveSMSEnable()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonReceiveSMSEnable(HRMONCON hCon, int enable)`

**Description** Using this function, it is possible to either enable or disable reception of the "fake" SMS messages from the RTCU unit by the function. Note, if disabling, the `rmonReceiveSMS()` will still block, waiting for a message to arrive.

**Input**

hCon	Handle to connection
enable	0=disable, 1=enable

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

## rmonGetGSMSignalLevel()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis:** `rmonRet __stdcall rmonGetGSMSignalLevel(HRMONCON hCon, int* signal)`

**Description** This functions check whether the GSM module in the RTCU unit currently is connected to a GSM base station (logged into the GSM network).  
 (This function does the same as the VPL function `gsmConnected()`).

**Input**

hCon	Handle to connection
------	----------------------

**Output**

signal	The GSM signal strength or 0 (zero) if not connected.
--------	---

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

## rmonSetAllowedCallerList()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonSetAllowedCallerList (HRMONCON hCon, const char numbers[81])`

**Description** Sets list of allowed phone numbers that can make incoming data calls to the RTCU. Phone numbers must be separated with the “,” character.  
 The list of allowed callers can also be set from the RTCU-IDE (menu: Unit -> Communication -> List of Callers).  
 (This function does the same as the VPL function `gsmSetListOfCallers()`).

**Input**

hCon	Handle to connection
numbers	List of phonenumbers separated by “,” character

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

## rmonGetAllowedCallerList()

**RTC model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonGetAllowedCallerList(HRMONCON hCon, char numbers[81])

**Description**                      Fetches list of allowed caller numbers set in rmonSetAllowedCallerList(). The list of allowed callers may also be fetched from the RTCU-IDE (menu: Unit -> Communication -> List of Callers). (This function does the same as the VPL function gsmGetListOfCallers()).

### Input

hCon	Handle to connection
------	----------------------

### Output

numbers	List of allowed caller numbers separated by “,” character.
---------	--

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

## rmonSetGSMPIN()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonSetGSMPIN (HRMONCON hCon, const char pin[5])`

**Description** Sets the SIM PIN code to use for the SIM card in the RTCU unit. This does NOT change the PIN code on the SIM card, it simply tells the RTCU unit 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.

**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 !**

Please notice the SIM PIN code may also be set using the RTCU IDE (menu: Unit -> Setup -> Set PIN code)  
 (This function does the same as the VPL function gsmSetPin()).

### Input

hCon	Handle to connection
pin	New SIM PIN code to be set

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

## rmonGetGSMPIN()

**RTC model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonGetGSMPIN (HRMONCON hCon, const char pin[5])`

**Description** Fetches the GSM SIM PIN code from the RTCU (see `rmonSetGSMPin()` above).  
 An empty string denotes that the PIN code has been disabled.

### Input

hCon	Handle to connection
------	----------------------

### Output

pin	Current SIM PIN code. Empty string specifies the PIN code to be currently disabled
-----	--

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

## Filesystem functions

This is a set of functions that offers a broad range of operations on the filesystem present in the RTCU unit. 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

The filesystem functions has these limitations in addition to the ones for the filesystem in general (See the RTCU-IDE Online-help/Manual):

1. A client can only have one open file at any given time. If more files are opened, the already open file is closed.
2. The working directory is shared with all other clients connected to the unit. Because of this it is recommended to use absolute paths where possible.

Note that the Filesystem API is only supported on the X32 generation of RTCU units.

## rmonMediaPresent()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonMediaPresent(HRMONCON hCon, int media, int* state, int* fserr)`

**Description** Queries whether the Media is present or not.

### Input

hCon	Handle to connection
media	The media ID. (0 = SD-CARD, 1 = Internal drive)

### Output

State	=0 (zero) if media is not present. <>0 (zero) if media is present
Fserr	Error code from the filesystem

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

## rmonMediaWriteprotected()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonMediaWriteprotected (HRMONCON hCon, int media, int* state, int* fserr)`

**Description** Queries whether the Media is write protected or not.

### Input

hCon	Handle to connection
media	The media ID. (0 = SD-CARD, 1 = Internal drive)

### Output

State	=0 (zero) if media is not write protected. <>0 (zero) if media is write protected.
Fserr	Error code from the filesystem

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

## rmonMediaOpen()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonMediaopen (HRMONCON hCon, int media, int\* fserr)

**Description**                 Open the media for use with the filesystem.

**Input**

hCon	Handle to connection
media	The media ID. (0 = SD-CARD, 1 = Internal drive)

**Output**

Fserr	Error code from the filesystem
-------	--------------------------------

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

## rmonMediaClose()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonMediaopen (HRMONCON hCon, int media, int\* fserr)

**Description**                 Close the media for use with the filesystem.

**Input**

hCon	Handle to connection
media	The media ID. (0 = SD-CARD, 1 = Internal drive)

**Output**

Fserr	Error code from the filesystem
-------	--------------------------------

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

## rmonMediaQuickformat()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonMediaQuickformat (HRMONCON hCon, int media, int* fserr)`

**Description** Quick formats the media.

### Input

hCon	Handle to connection
media	The media ID. (0 = SD-CARD, 1 = Internal drive)

### Output

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## rmonMediaEject()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonMediaEject(HRMONCON hCon, int media, int* fserr)`

**Description** Eject the media.

### Input

hCon	Handle to connection
media	The media ID. (0 = SD-CARD, 1 = Internal drive)

### Output

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## rmonMediaInformation()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonMediaInformation(HRMONCON hCon, struct rmonMediaInfo media[8])`

**Description** Queries the unit for which media is mounted, what type of media it is, and the capacity of the media.  
 The media type can be one of the following:  
 0 = Not mounted, 1 = SD-CARD, 2 = Internal FLASH.

### Input

hCon	Handle to connection
------	----------------------

### Output

media	An array of media information structures.
-------	---

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

```
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
};
```

## rmonFSStatusLED()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFSStatusLED(HRMONCON hCon, int* fserr)`

**Description** Using this function, it is possible to either enable or disable the filesystem status LED's.

### Input

hCon	Handle to connection
Enable	0=disable, 1=enable

### Output

Fserr	Error code from the filesystem
-------	--------------------------------

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

## rmonDirCreate()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonDirCreate(HRMONCON hCon, const char name[61], int* fserr)`

**Description** Create a new directory.

### Input

hCon	Handle to connection
Name	Name of the directory to create. (60 characters + 0 (zero) terminator) Both absolute and relative path can be used.

### Output

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## rmonDirChange()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonDirChange(HRMONCON hCon, const char path[61], int\* fserr)

**Description**                      Change the working directory.

**Input**

hCon	Handle to connection
Path	Path to the new working directory. (60 characters + 0 (zero) terminator) Both absolute and relative path can be used.

**Output**

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## rmonDirCurrent()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                              rmonRet \_\_stdcall rmonDirCurrent(HRMONCON hCon, char path[61], int\* fserr)

**Description**                              Retrieve the absolute path to the working directory.

**Input**

hCon	Handle to connection
------	----------------------

**Output**

Path	The absolute path to the working directory. (60 characters + 0 (zero) terminator)
Fserr	Error code from the file system.

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

## rmonDirCatalog()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonDirCatalog(HRMONCON hCon, short index, char name[15], struct rmonRTCTime* time, long* length, int* fserr)`

**Description** Retrieves the information of an entry in the working directory.

### Input

hCon	Handle to connection
Index	The index of the directory entry to retrieve.

### Output

Name	The name of the directory entry. (14 characters + zero terminator)
Time	The creation time of the file. Uses the same structure as rmonGetRTC. Not used for directories.
Length	The size of the file. Not used for directories.
Fserr	Error code from the filesystem.

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

## rmonDirDelete()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonDirDelete(HRMONCON hCon, const char name[61], int* fserr)`

**Description** Delete a directory.

### Input

hCon	Handle to connection
Name	The name of the directory. (60 characters + zero terminator) Both absolute and relative path can be used.

### Output

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## rmonFileCreate()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonFileCreate(HRMONCON hCon, const char name[61], int\* fserr)

**Description**                      Creates a new file. If a file is already open, it will be closed before the new file is created.

**Input**

hCon	Handle to connection
Name	The name of the file to create. (60 characters + zero terminator) Both absolute and relative path can be used.

**Output**

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## rmonFileOpen()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                              rmonRet \_\_stdcall rmonFileOpen(HRMONCON hCon, const char name[61], int\* fserr)

**Description**                              Opens a file. If a file is already open, it will be closed before the new file is opened.

**Input**

HCon	Handle to connection
Name	The name of the file to create. (60 characters + zero terminator) Both absolute and relative path can be used.

**Output**

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## rmonFileExists()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFileExists(HRMONCON hCon, const char name[61], char* state, int* fserr)`

**Description** Query whether a file exists.

### Input

hCon	Handle to connection
Name	The name of the file to create. (60 characters + zero terminator) Both absolute and relative path can be used.

### Output

State	=0 (zero) if file does not exist. <>0 (zero) if file does exist.
Fserr	Error code from the filesystem.

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

## rmonFileRename()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFileRename(HRMONCON hCon, const char name_old[61], const char name_new[13], int* fserr)`

**Description** Renames a file

### Input

hCon	Handle to connection
Name_new	The new name of the file. (12 characters + zero terminator)
Name_old	The name of the file to rename. Both absolute and relative paths can be used (60 characters + zero terminator)

### Output

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## rmonFileDelete()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonFileDelete(HRMONCON hCon, const char name[61], int\* fserr)

**Description**                      Delete a file.

**Input**

hCon	Handle to connection
Name	The name of the file to create. (60 characters + zero terminator) Both absolute and relative path can be used.

**Output**

State	=0 (zero) if file does not exist. <>0 (zero) if file does exist.
Fserr	Error code from the filesystem.

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

## rmonFileStatus()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                              rmonRet \_\_stdcall rmonFileStatus(HRMONCON hCon, int\* status, int\* fserr)

**Description**                      Retrieve the status of the open file.

**Input**

hCon	Handle to connection
------	----------------------

**Output**

Status	The status of the file. Identical to the return value of the fsFileStatus VPL function.
Fserr	Error code from the filesystem.

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

## rmonFileGetInfo()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFileGetInfo(HRMONCON hCon, const char name[61], struct rmonRTCTime* time, long* length, int* fserr)`

**Description** Retrieve the size and creation timestamp of a file.

### Input

hCon	Handle to connection
name	The name of the file. (60 characters + zero terminator) Both absolute and relative path can be used.

### Output

Time	The creation timestamp. Please refer to the definition of rmonRTCTime above (Page 40)
Length	The size of the file in bytes.
Fserr	Error code from the filesystem.

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

## rmonFileSeek()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFileSeek(HRMONCON hCon, long offset, int* fserr)`

**Description** Moves the file pointer.

### Input

hCon	Handle to connection
Offset	The new position relative to the Start of file. >0 (zero) – Position in file. =0 (zero) – Start of file. -1 – End of file.

### Output

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## rmonFilePosition()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFilePosition(HRMONCON hCon, long* position, int* fserr)`

**Description** Retrieve the file pointer position of the open file.

**Input**

hCon	Handle to connection
------	----------------------

**Output**

Position	The file pointer position
Fserr	Error code from the filesystem.

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

## rmonFileRead()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFileRead(HRMONCON hCon, int elemcnt, char* buffer, int* elemread, int* fserr, rmoncbprogress pfunc, void* uptr)`

**Description** Read a block of data from file.

**Input**

hCon	Handle to connection
Elemcnt	The number of bytes to read from file.
Pfunc	Pointer to function where progress is reported.
Arg	Pointer to user argument used when reporting progress.

**Output**

Buffer	The buffer where the data read from the file is stored.
elemread	The number of bytes read from file.
Fserr	Error code from the filesystem.

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

## rmonFileReadString()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFileReadString(HRMONCON hCon, char str[241], int* elemread, int* fserr)`

**Description** 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).

### Input

hCon	Handle to connection
------	----------------------

### Output

str	The buffer where the string read from the file is stored. (240 characters + zero terminator)
elemread	The number of bytes read from file.
Fserr	Error code from the filesystem.

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

## rmonFileWrite()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFileWrite(HRMONCON hCon, int elemcnt, char* buffer, int* elemwr, int* fserr, rmoncbprogress pfunc, void* uptr)`

**Description** Write a block of data to file.

### Input

hCon	Handle to connection
elemcnt	The number of bytes to write to file.
Buffer	The buffer where the data to write is stored.
pfunc	Pointer to function where progress is reported.
uptr	Pointer to user argument used when reporting progress.

### Output

elemwr	The number of bytes written to file
Fserr	Error code from the filesystem.

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

## rmonFileWriteString()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFileWriteString(HRMONCON hCon, const char str[241], int* elemwr, int* fserr)`

**Description** Write a string to file.

### Input

hCon	Handle to connection
str	The string to write to file. (240 characters + zero terminator)

### Output

elemwr	The number of bytes written to file
Fserr	Error code from the filesystem.

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

## rmonFileWriteStringNL()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFileWriteStringNL(HRMONCON hCon, const char str[241], int* elemwr, int* fserr)`

**Description** Write a string to file. <CR><LF> are appended.

### Input

hCon	Handle to connection
str	The string to write to file. (240 characters + zero terminator)

### Output

elemwr	The number of bytes written to file
Fserr	Error code from the filesystem.

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

## rmonFileClose()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonFileClose(HRMONCON hCon, int\* fserr)

**Description**                Close the file.

**Input**

hCon	Handle to connection
------	----------------------

**Output**

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## rmonFileFlush()

**RTCU model:** X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonFileFlush(HRMONCON hCon, int\* fserr)

**Description**                Flush cached write operations to media.

**Input**

hCon	Handle to connection
------	----------------------

**Output**

Fserr	Error code from the filesystem.
-------	---------------------------------

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

## Misc. functions

Below is a list of different “housekeeping” functions.

### rmonReset()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                    rmonRet \_\_stdcall rmonReset(HRMONCON hCon)

**Description**                This function will reset the connected RTCU unit. If the RTCU unit is remotely connected (modem or RTCU gateway) the reset will be delayed until the connection is lost (by calling rmonDisconnect() etc). However, if the connection is thru a direct cable connection, the RTCU unit executes the reset command immediately. The reset has the same effect as cycling power to the RTCU unit, the VPL program starts executing from the start again. This can also be carried out from the RTCU-IDE (menu: Unit -> Execution -> Reset)  
 (This function does the same as the VPL function boardReset()).

**Input**

hCon	Handle to connection
------	----------------------

**Returns**                    rmonOK, rmonComError, rmonIllegalHandle

### rmonHalt()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                    rmonRet \_\_stdcall rmonHalt(HRMONCON hCon)

**Description**                Stops the currently executing VPL program in the RTCU.  
 This can also be carried out from the RTCU-IDE (menu: Unit -> Execution -> Halt)  
 The RTCU unit can be started again with the reset command (see above) or by cycling power.

**Input**

hCon	Handle to connection
------	----------------------

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

## rmonGetSerialNumber()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonGetSerialNumber(HRMONCON hCon, unsigned long *SerialNumber)`

**Description** Returns the serial number of the connected RTCU.

**Input**

hCon	Handle to connection
------	----------------------

**Output**

SerialNumber	The serialnumber of the RTCU unit.
--------------	------------------------------------

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

## rmonIsUnitProgrammable()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonIsUnitProgrammable(HRMONCON hCon, unsigned char *Programmable)`

**Description** Checks if the connected RTCU is programmable (a MAX unit) or not programmable (a MICRO unit)

**Input**

hCon	Handle to connection
------	----------------------

**Output**

Programmable	1 if unit is programmable (a MAX unit), 0 if not programmable (a MICRO unit)
--------------	--

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

## rmonVer()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonVer(HRMONCON hCon, int *ver)`

**Description** Returns the Firmware version of the connected RTCU.  
 Note that this function must be used to determine if RTCU unit is in monitor mode.

**Input**

hCon	Handle to connection
------	----------------------

**Output**

ver	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.
-----	--

**Returns** `rmonOK, rmonComError, rmonIllegalHandle`

## rmonSetPassword()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonSetPassword (HRMONCON hCon, const char password[21])`

**Description** 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-IDE (menu: Unit -> Setup -> Set Password)

**Input**

hCon	Handle to connection
password	New password to be set (zero terminated ASCII String).

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

## rmonGetTargetInfo()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonGetTargetInfo(HRMONCON hCon, int* targetID, int* firmwareVer);`

**Description** Fetches RTCU type and firmware version from the RTCU.

**Input**

hCon	Handle to connection
------	----------------------

**Output**

targetID	Please see the table below for a list of possible target ID's.
firmwareVer	Firmware version always different from 0 and scaled by 100 (Version 4.66 is returned as 466)

**Returns** `rmonOK, rmonComError, rmonIllegalHandle`

Symbolic name	Value	Description
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_ICP104	104	RTCU-DX4 Series
RMONTGT_ICP109	109	RTCU-AX9 Series
RMONTGT_ICP192	192	RTCU-MX2 SOM.

## rmonGetTargetProfile()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonGetTargetProfile(HRMONCON hCon, int* targetID, int* firmwareVer, unsigned long* SerialNumber);`

**Description** Fetches RTCU type, serial number and firmware version from the RTCU.

### Input

hCon	Handle to connection
------	----------------------

### Output

targetID	Please see the table below for a list of possible target ID's.
firmwareVer	Firmware version always different from 0 and scaled by 100 (Version 4.66 is returned as 466)
SerialNumber	The serialnumber of the RTCU unit.

**Returns** `rmonOK, rmonComError, rmonIllegalHandle`

Symbolic name	Value	Description
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_ICP104	104	RTCU-DX4 Series
RMONTGT_ICP109	109	RTCU-AX9 Series
RMONTGT_ICP192	192	RTCU-MX2 SOM.

## rmonReceiveDebugMsg()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonReceiveDebugMsg (HRMONCON hCon, char* msg, int maxsize)`

**Description** Receive any incoming Debug messages from RTCU.  
 Please notice that rmonReceiveDebugMsg blocks and will not return before a debug message has been received (see also rmonReceiveSMS()).

### Input

hCon	Handle to connection
maxsize	Maximum number of characters to receive

### Output

msg	Buffer with received debug message
-----	------------------------------------

**Returns** rmonOK, rmonIllegalHandle, rmonNoData

## rmonGetDebugEnable()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonGetDebugEnable(HRMONCON hCon, int* enabled )`

**Description** Checks if Debug messages has been enabled or disabled in unit.

### Input

hCon	Handle to connection
------	----------------------

### Output

enabled	1 if Debug messages is enabled, 0 if Debug messages is disabled
---------	---

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

## rmonSetDebugEnabled()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonSetDebugEnabled(HRMONCON hCon, int enabled )

**Description**                Enable or disable Debug messages from unit.

**Input**

hCon	Handle to connection
enabled	1 to enable Debug messages, 0 to disable Debug messages

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

## rmonVoiceMessagesAbove64K()

**RTCU model:** Large  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonVoiceMessagesAbove64K (HRMONCON hCon, char \*above)

**Description**                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.

**Input**

hCon	Handle to connection
------	----------------------

**Output**

above	1 if Voice messages above 64k, 0 if no Voice messages above 64k
-------	---

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

## rmonGetAppInfo()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonGetAppInfo(HRMONCON hCon, char \*Appname, int \*Appver )

**Description**                Fetches Application name and version from the RTCU.

### Input

hCon	Handle to connection
------	----------------------

### Output

Appname	0 (zero) terminated string containing the application name. Max. 15 characters long.
Appver	Application version scaled by 100 (Version 4.66 is returned as 466)

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

## rmonGetGPRSSettings()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonGetGPRSSettings(HRMONCON hCon, rmonGPRSSettings* Settings);`

**Description** This function fetches the TCP/IP settings the unit uses to connect over GPRS. The GPRS settings retrieved from the RTCU unit are identical to those retrieved with the “Fetch from RTCU” button in the RTCU IDE (Unit->GPRS->TCP/IP settings).

All the general TCP/IP parameters use a binary packed IP address (a.b.c.d) using this format:

a	bit 24..31
b	bit 16..23
c	bit 8..15
d	bit 0..7

### Input

hCon	Handle to connection
------	----------------------

### Output

Settings	A structure containing the GPRS settings
----------	--

**Returns** `rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonNoData`

```
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;
```

## rmonSetGPRSSettings()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonSetGPRSSettings(HRMONCON hCon, rmonGPRSSettings Settings);`

**Description** This function sets the TCP/IP settings the unit uses to connect over GPRS. This function is identical to the VPL function sockSetTCPIPParam, and the TCP/IP settings dialog (Unit->GPRS->TCP/IP settings) in the RTCU IDE.

All the general TCP/IP parameters use a binary packed IP address (a.b.c.d) using this format:

- a      bit 24..31
- b      bit 16..23
- c      bit 8..15
- d      bit 0..7

### Input

hCon	Handle to connection
Settings	A structure containing the GPRS settings

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

```
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;
```

## rmonGetGatewaySettings()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonGetGatewaySettings(HRMONCON hCon, rmonGWSettings* Settings);`

**Description** This function fetches the settings the unit uses to connect to Gateway. The Gateway settings retrieved from the RTCU unit are identical to those retrieved in the RTCU IDE with the “Fetch from RTCU” button in the Gateway settings dialog (Unit->GPRS->Gateway settings).

### Input

hCon	Handle to connection
------	----------------------

### Output

Settings	A structure containing the Gateway settings
----------	---

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

```
typedef struct {
    // RTCU GPRS Gateway 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;
```

## rmonSetGatewaySettings()

**RTCU model:** Large & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonSetGatewaySettings(HRMONCON hCon, rmonGWSettings Settings);`

**Description** This function fetches the settings the unit uses to connect to Gateway. This function is identical to the VPL function sockSetGWParam, and the settings are identical to those written from the RTCU IDE with the "Write to RTCU" button in the Gateway settings dialog (Unit->GPRS->Gateway settings).

### Input

hCon	Handle to connection
Settings	A structure containing the Gateway settings

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

```
typedef struct {
    // RTCU GPRS Gateway 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;
```

## rmonFaultLogRead()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFaultLogRead(HRMONCON hCon, rmonFault *Fault)`

**Description** Fetches the Fault log from the RTCU.  
 This function is identical to the fetch button in the fault log in the RTCU IDE

### Input

hCon	Handle to connection
------	----------------------

### Output

Fault	The function fills this structure with the fault log entries.
-------	---

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

```
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;
    rmonFaultRecord  Record[32];
} rmonFault;
```

## rmonFaultLogClear()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFaultLogClear(HRMONCON hCon)`

**Description** This function clears the fault log.  
 This function is identical to the clear button in the Fault log in the RTCU IDE.

**Input**

hCon	Handle to connection
------	----------------------

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

## rmonFaultLogGetText()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis** `rmonRet __stdcall rmonFaultLogGetText(HRMONCON hCon, char* FaultText, int bufsize)`

**Description** This function retrieves the text message for a Fault code.

**Input**

hCon	Handle to connection
bufsize	The size of the buffer where the text is stored

**Output**

FaultText	0 (zero) terminated string containing the fault message
-----------	---

**Returns** `rmonOK`

## rmonSoftwareUpgrade()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
 Connected State

**Synopsis**                      rmonRet \_\_stdcall rmonSoftwareUpgrade(HRMONCON hCon, char string[35], int \*res)

**Description**                      Upgrades the RTCU unit.

**Input**

hCon	Handle to connection
String	0 (zero) terminated string. The upgrade key.

**Output**

res	The type of upgrade performed.
-----	--------------------------------

**Returns**                              rmonOK, rmonComError, rmonIllegalHandle, rmonTargetError, rmonError, rmonIllegalTarget

Value	Description
0	Not upgraded / Wrong upgrade key
1	GPRS enabled
2	Unit 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

## rmonFlexOption()

**RTCU model:** Large, Small & X32  
**Called in:** Locally & Remotely  
Connected State

**Synopsis** rmonRet \_\_stdcall rmonFlexOption(HRMONCON hCon, char string[12])

**Description** This function will enable certain options in the unit.

### Input

hCon	Handle to connection
string	Zero terminated string. The option key.

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

## 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 unit
        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 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 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 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;

    // Open RTCUCSP library
    rmonOpen();

    // Open a connection
    hCon = rmonOpenConnection();

    // Select which comports to use for local and remote connections. (Use COM0 if no port is to be used)
    rmonSetComport(hCon, "COM1", "COM0");

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

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

    // Wait for a connection to a RTCU unit
    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 unit.\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 unit !\n");
        return 1;
    }

    // Get information about the unit
    int targetid, firmwareversion;
    rmonGetTargetInfo(hCon, &targetid, &firmwareversion);
    printf("Target ID=%i, Firmware version=%i.%02i\n", targetid, firmwareversion/100,
firmwareversion%100);

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

    // Use this to upload new firmware to the unit:
    //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);
    */
}
```

```
// Close connection after use
rmonCloseConnection(hCon);

printf("\n\nPress any key to end program...\n\n");
while (true) {
    if (getch())
        break;
}

return 0;
}
```

## Appendix B, RTCUPROG application

The RTCUPROG 6.12 program is a complete Microsoft Visual Studio C++ 2005 project. This program demonstrates all aspects in making a robust application that will manage the connection to both local and remote RTCU unit. The application allows the user to upload new firmware to a unit, or upload a complete new project to the RTCU unit, including both VPL program and voice messages.