CONTROLTOOLBOX.COM

Smart Modbus Server For Smart Device

Developer's Manual

12/26/2008

Connect LAN/WAN devices to SCADA via Modbus TCP

NetPro-ModServLiteCE Developer's Manual

Smart Modbus Server
For Smart Device

Version 1.00
Last update: December 2008

A PRODUCT OF WWW.CONTROLTOOLBOX.COM

Controltoolbox.com is owned by:
Avista Realtime Systems, LLC
Tel: 410 312 5590
Fax: 866 897 2842
Address:
Avista Realtime Systems
10015 Old Columbia Rd
Suite B-215
Columbia, MD 21046
Web: www.avistarealtime.com
Technical Support:
support@controltoolbox.com

The software described in this manual is furnished under a license agreement and may be used only in accordance with the terms of that agreement.

Disclaimer
Information in this document is subject to change without notice and does not represent a commitment on the part of Avista Realtime Systems or ControlToolBox.

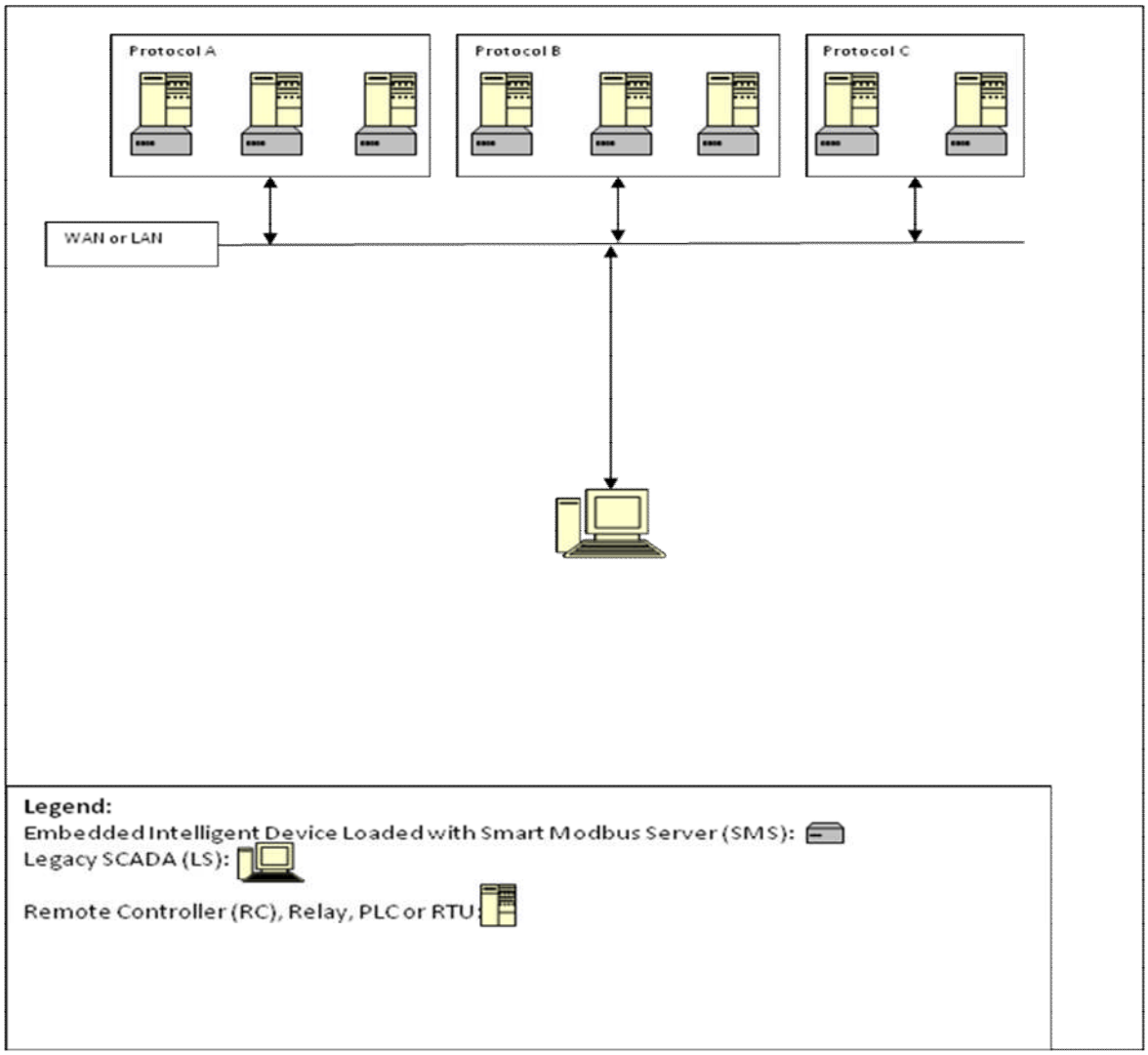
This document is provided "as is," without warranty of any kind, either expressed or implied, including, but not limited to, its particular purpose. Rights to make improvements and/or changes to this manual, or to the products and/or the programs described in this manual, are reserved.

Information provided in this manual is intended to be accurate and reliable. However, Avista assumes no responsibility for its use, or for any infringements on the rights of third parties that may result from its use. This product might include unintentional technical or typographical errors. Changes are made periodically to the information in this manual to correct such errors, and these changes are incorporated into new editions of the publication.

1. Introduction

NetPro-ModServLiteCE is a Smart Modbus TCP Server (SMS). The product is a set of libraries developed under .Net Framework to allow the developers to build their own Modbus TCP Server in almost no time. The library allows the server to emulate a large number Modbus TCP RTUs. This is good for situations that developer needs to build a Modbus RTU, data concentrator or an RTU with different profiles.

The following figure demonstrates a sample application for the server:



The data is collected at SMS where it is presented as one or multiple Modbus TCP RTUs to the SCADA or control systems. Data is collected via the device own I/Os or through device's serial port where it acts as a master. SMS passes the field data to the Modbus client while taking care of communication to field devices. SMS can emulate up to 100 Modbus TCP RTUs.

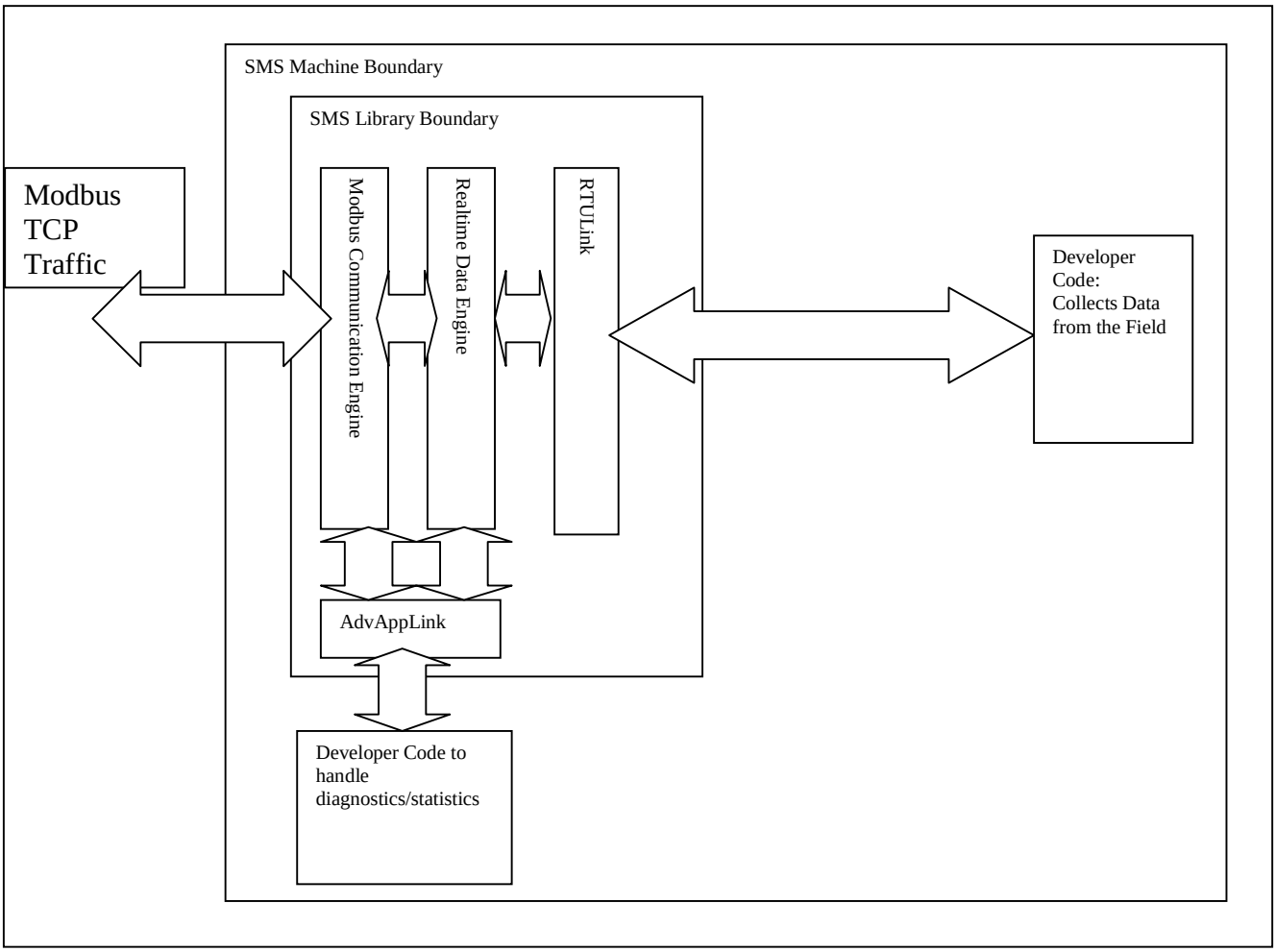
In addition to the above architecture, SMS may be used for the following applications:

- Smart grid applications when a large distributed community of devices needs to communicate to the central.
- Building Automation.
- Ideal for Water, Electric, Transportation, Gas and Oil SCADA that deal with a distributed system.
- Metering applications.
- Build your own I/O Server
- Protocol Converter
- Modbus TCP RTU Simulator
- Build your own Modbus TCP RTU product
- Data Statistician and Sequence of Event (SoE) logger
- Smart gateway to block unwanted controls or commands

2. Code Architecture

The code can run on Windows Device Operating Systems such as Windows CE. This code is written based on the latest .Net Compact Framework (3.5). Therefore, it is necessary to install the Framework before trying to run the applications.

The following figure demonstrates the code architecture and internal interaction between different modules:



2.1 Modbus Communication Engine Module

This module is responsible of opening/closing the listener TCP ports for Modbus traffic, accepting the incoming connect calls and managing Modbus traffic. In case of advanced applications option, this module collects change of value and sequence of event information about the Modbus traffic per RTU.

2.2 Real-time Data Engine Module

This module is responsible of creating and destroying RTUs in the server database. RTUs are created or destroyed by the incoming requests from the user code. In addition to managing the RTUs, the module keeps the data values and fires events to the user code to push the control commands. In case of advanced applications, real-time data engine reports sequence of event and change of value information to the advanced application interface.

2.3 RTULink Module

RTULink is the module that is responsible to communicate the server information to the user code and vice versa. *RTULink* exposes APIs that can be called to access the server information. Once client application connects to the server, it obtains a pointer to an internal object that is built based on *IRTULink*. *RTULink* is inherited from *IRTULink*. As will be seen in code examples, communication to the server from this point on is seamless as though the real-time data engine is accessed directly by the user application.

2.4 AdvAppLink Module

AdvAppLink is the module that is responsible to communicate the server statistics/diagnostics information to the user application. *AdvAppLink* exposes APIs that can be called from the user application to access the server information. The user application needs to obtain a pointer to this class that is built based on *IAdvAppLink*. *AdvAppLink* is inherited from *IAdvAppLink* . As will be seen in code examples, communication to the server from this point on is seamless as the real-time data engine is accessed directly by user application.

3. Building the Server

Building the server code is easy and fast. Depending on how you would like to package the code, you need to create an appropriate new C# project. If you are intending to package the code in a stand-alone application, you need to open a particular type of Smart Device application.

Once the application is opened, add the following references from the install directory to the application:

- *ARTS_MODIntf.dll*: This library includes all the definitions necessary for communication between user code and the server.
- *NetProModServLiteCE.dll*: This library provides the definitions for the server module.

Also, please make sure to include the following statements in your header to make reference to the libraries we just mentioned:

```
using ARTS_MODIntf;
using ARTS_MODLiteCE;
```

After adding the references, you need to include an instance of the main class *NetProModServLiteCE* in your code and simply call to process it with the frequency you wish. Please note that even though *NetProModServLiteCE* is a very powerful class and takes care of all Modbus and remote device communication, its interface to the program is very simple. Developer needs to instantiate it, call it to process and shut it down when it is not needed any more.
Here are the steps that you need to follow in details:

3.1 Instantiate

First, you need to instantiate the class. The constructor has the following signature:

```
public NetProModServLiteCE(int _modbusTCPPort, int _noofRTUs, bool _advancedApplications,
        string _clientID, string _license, out bool _licenseValid)
```

Where:

_ *modbusTCPPort* is the port you wish to listen for incoming Modbus calls. It is typically set to 502.
_noofRTUs is the number of RTUs that you intend to cover with the server. Please note that increasing the number of RTUs may require an upgrade in your license.
_advancedApplications shows if you are allowed to use advanced applications. Please note that using advanced applications may require upgrading your license.
_clientID is the reference number assigned to the client.
_license is the license number generated for the client.
_licenseValid is an output parameter that returns true if the licensing engine is able to validate the license. If the license cannot be validated, the module works for 30 minutes for your evaluation.

Once you successfully instantiate the class, it is ready to process the incoming Modbus calls. Please note that the Modbus RTUs are created dynamically in the server by the user code.

3.2 Process

You need to simply call the *Process()* method of the class that you just created on a continuous basis. You may want to call this segment of the code in a separate thread. The faster you call this process, the faster your Modbus incoming calls are processed. However, if you do not expect a very fast turnaround time, you can call the method with lower speed.

3.3 Shutdown

Once you are done with the library and do not need further Modus processing, you need to call *ShutDown()* method to release the servers resources. However, in most scenarios the code runs indefinitely.

3.4 Sample Code

Here is a code segment that shows how you can get your server up and running:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using ARTS_MODIntf;
using ARTS_ModLiteCE;
using System.Threading;

namespace NetProModServLiteCE_test
{
   static class Program
   {
      static void Main()
      {
         bool licenseValid;
         // we are calling the server with no valid license
         // it should work for 30 minutes
         NetProModServLiteCE modServ = new NetProModServLiteCE(502, 1, true, "None",
            "None", out licenseValid);

         //service loop
         bool runFlag = true;
         while (runFlag)
         {
            // process the server
            modServ.Process();

            // wait a bit
            // you may need to decrease the time here if you need faster response to modbus
            // it can go down to 5 milliseconds
            Thread.Sleep(100);
```

```
            }

            //shutdown if you get somehowe out of the previous loop!
            modServ.ShutDown();
        }
    }
}
```

4. Building the User Code

Building the user is as easy as building the server. All the licensing issues are taken care of at the server level. Here are the steps user needs to follow:

- Get an RTU link object to connect to the server *RTULink* class.
- Create the RTU in the server domain space.
- Determine size of the RTU database to specify how many points of each object (coil, discrete input, input register and holding register) it has.
- Subscribe for the events and the type of events.
- Initialize the data point values at the server.
- Enable the RTU.
- Process the events fired from the server.

4.1 Get RTU Link Object

RTU Link object creates a link between the *RTULink* class in the server and the client application. Here is a sample code for creating this object:

```
IRTULink rtuLink = modServ.CreateRTULink();
```

4.2 Create the RTU

In this step, we create an RTU in the server database. The following is a segment of the code that creates the RTU:

```
rtuLink.Ctreate(unitID, out rtuKey);
```

rtuKey needs to be defined as *Guid*. This is the key used in subsequent calls to identify the RTU to the server.

4.3 Size RTU Database

Client can determine the size of the RTU database created in the server. Here is a sample code to demonstrate the operation:

```
/* set discrete input data size */
rtuLink.SetDatabaseSize(rtuKey, GeneralDef.DataType.DiscreteInput, 16);

/* set coil data size */
rtuLink.SetDatabaseSize(rtuKey, GeneralDef.DataType.Coil, 16);

/* set input register data size */
rtuLink.SetDatabaseSize(rtuKey, GeneralDef.DataType.InputRegister, 16);

/* set holding register data size */
```

```
rtuLink.SetDatabaseSize(rtuKey, GeneralDef.DataType.HoldingRegister, 16);
```

In this example, discrete input, coil, input register and holding register databases are all set to 16 points.

4.4 Subscribe for the Events

Client can subscribe to the server for the events of the points of interest. Events can be registered in two ways:

- Receive events based on any write operation generated by the Modbus communication.
- Receive event only if the value of the point has been changed in the server because of a Modbus data communication.

The following sample code demonstrates how the events can be defined:

```
GeneralDef.ElementGroup coilEventStatusGroup = new GeneralDef.ElementGroup();
GeneralDef.ResultGroup coilResult;

/* set events for coils */
for (int c = 0; c < 16; c++)
{
   GeneralDef.EventStatusElement eventStatusElement =
      new GeneralDef.EventStatusElement();
   eventStatusElement.dataType = GeneralDef.DataType.Coil;
   eventStatusElement.eventType = GeneralDef.EventType.AnyWrite;
   eventStatusElement.address = c;

   /* add the element to the list */
   coilEventStatusGroup.list.Add(eventStatusElement);
}
result = rtuLink.SetGroupEventStatus(rtuKey, coilEventStatusGroup,
   out coilResult);

GeneralDef.ElementGroup hrEventStatusGroup = new GeneralDef.ElementGroup();
GeneralDef.ResultGroup hrResult;

/* set events for holdong registers */
for (int c = 0; c < 16; c++)
{
   GeneralDef.EventStatusElement eventStatusElement =
      new GeneralDef.EventStatusElement();
   eventStatusElement.dataType = GeneralDef.DataType.HoldingRegister;
   eventStatusElement.eventType = GeneralDef.EventType.OnChange;
   eventStatusElement.address = c;

   /* add the element to the list */
   hrEventStatusGroup.list.Add(eventStatusElement);
}
result = this.rtuLink.SetGroupEventStatus(this.rtuKey, hrEventStatusGroup,
   out hrResult);
```

The first loop subscribes events for a group of coil points for any write operation generated by Modbus. The second loop subscribes a group of holding register points only for the events that are generated by change of value caused by Modbus data communication at the server level.

4.5 Initialize Data Points

Once the desired events are registered, client needs to initialize the point values at the server. Please note that so far, we have not enabled the RTU in the server. This means that the server is not responding to the incoming Modbus polls yet. Here is a sample code that shows how to write input register values into the server database:

```
/* discrete input */
GeneralDef.ElementGroup diData = new GeneralDef.ElementGroup();
GeneralDef.ResultGroup diResult;
for (int i = 0; i <16; i++)
{
   /* initialize the write element */
   GeneralDef.WriteElement writeElement = new GeneralDef.WriteElement();
   writeElement.dataType = GeneralDef.DataType.DiscreteInput;
   writeElement.value = i % 2;
   writeElement.address = i;

   /* add the element to the list */
   diData.list.Add(writeElement);
}
result = .rtuLink.WriteGroupData(rtuKey, diData, out diResult);
```

Here input registers with odd indexes are set to one and the ones with even indexes are set to zero. The code for writing other data types is the very similar. The difference is where the write element data type is set.

4.6 Enable the RTU

At this point, all the points are initialized and the desired events are registered. However, the server is still not responding to the Modbus commands. We need to enable the RTU in order to allow the server to respond to Modbus polls for this particular RTU. Here is how the RTU is enabled:

```
rtuLink.Enable(rtuKey, true);
```

4.7 Process the Events

Once the RTU is enabled, it will start queuing the events generated by Modbus communication for each RTU. Events are generated based on the previous registration set by the client. Client needs to call the event function to receive its queued events. The event function has a wait argument that shows how long the process needs to wait at the server before getting back to the client code. The following code shows how the function is used:

```
for (; ; )
```

```
{
    GeneralDef.DataType _type;
    int _address;
    int _value;

    /* get the events */
    if (rtuLinl.PopEvent(_rtuKey, out _type, out _address,  out _value, TimeSpan.FromSecond(30)))
        MessageBox.Show("event: type: " + _type.ToString() +
            ", value: " + _value.ToString() +
            ", address: " + _address.ToString());

    /* wait a bit */
    Thread.Sleep(100);
}
```

5. Advanced Applications

Advanced applications are a set of applications provided at the server level. The applications allow the developer to access some very important information about how the system behaves over time. As mentioned earlier, advanced applications are implemented in the server *AdvAppLink* module. The user application needs to get access to this module via an object of *IAdvAppLink*. This object allows the user to read or clear the information. The information provided by advanced applications is as follows:

- Number of change of status per point/per RTU
- Sequence of Event per point/per RTU
- Number of calls per function call/per RTU
- Sequence of calls plus the exact message string per function call/per RTU
- List of connected clients to the server
- Block/unblock specific controls for safety purposes

In addition to reading the counters, the user application is allowed to clear the counters as well. Sequence of event table is cleared every time read.

The following code demonstrates how the advanced application object is created:

```
/* get the advance application link */
IAdvAppLink advAppLink = modbusServer.GetAdvAppLink();
```

modbusServer is the server object that is created in the server application (Please refer to the Server Build section).

For more information about how to retrieve the advanced information, please refer to the section where the advanced application APIs are explained. The following example shows how the function code 5 counter is retrieved:

```
/* get advanced info of funtion code 5 */
uint msgNo;
bool rollOver;
DateTime startTime;
advAppLink.ReadFCMsgNo(1, 5, out msgNo, out rollOver, out startTime);
```

Here, *msgNo* is the number of function code 5 commands from the start time, *rollOver* is a flag that shows if the counter has had an overflow and *startTime* shows the time that data collection started. Reviewing the APIs, you will see that the data collection can be reset. That in turn resets the start time.

6. Supported Functions

The following table shows the function codes that are supported by the Modbus server:

Function Code	Description	Notes
01	Read Coil	Supported
02	Read Discrete Input	Supported
03	Read Holding Register	Supported
04	Read Input Register	Supported
05	Write Coil	Supported
06	Write Register	Supported
07	Read Exception Status	Not Supported, specific to serial communication
08	Diagnostics	Not Supported, specific to serial communication
11	Get Com Event Counter	Not Supported, specific to serial communication
12	Get Com Event Log	Not Supported, specific to serial communication
17	Report Slave ID	Not Supported, specific to serial communication
15	Write Multi Coil	Supported
16	Write Multi Register	Supported
20	Read File Record	Not Supported, will be implemented in the next release
21	Write File Record	Not Supported, will be implemented in the next release
22	Mask Write Register	Supported
23	Read/Write Multiple Register	Supported
24	Read FIFO Queue	Not Supported, will be implemented in the next release
43	Read Device Identification	Not Supported, will be implemented in the next release

7. ARTS_MODIntf Library

ARTS_MODIntf.dll includes all the common definition required by the server and clients. This library has the following main classes/interfaces: *GeneralDef*, *IRTULink*, *IAdvApp*.

7.1 GeneralDef

GeneralDef contains all the common data structure definitions between client code and the server.

7.1.1 GeneralDef.DataType

```
public enum DataType{
    DiscreteInput,
    Coil,
    InputRegister,
    HoldingRegister,
    Record,
    None}
```

This enumeration represents the different Modbus data types. *DiscreteInput* stands for Modbus discrete input data type, *Coil* stands for Modbus coil data type, *InputRegister* stands for Modbus input register data type, *HoldingRegister* stands for Modbus holding register data type, and *Record* stands for a file record.

7.1.2 GeneralDef.EventType

EventType:

```
public enum EventType
{
    OnChange,
    AnyWrite,
    None
}
```

This enumeration is used for specifying type of events when events are registered by the client in the server space. Here, *OnChange* means that the client is interested in event only when there is a change of value, *AnyWrite* means that the client is interested in any write operation caused by Modbus communication even though it does not generate change of value and *None* means that the remote client is not interested in events for the particular point.

7.1.3 GeneralDef.EventStatusElement

EventStatusElement:

```
public struct EventStatusElement
{
```

```
    public GeneralDef.DataType dataType;
    public int address;
    public GeneralDef.EventType eventType;
}
```

This structure is used to carry the information about event registration when client is registering events in the server space. *dataType* stands for the Modbus data type, *address* stands for point address and *eventType* stadns for the desired event type.

7.1.4 GeneralDef.WriteElement

WriteElement:

```
public struct WriteElement
{
    public GeneralDef.DataType dataType;
    public int address;
    public int value;
}
```

Whenever the client needs to write into the server database, it uses this structure to include the data it targets. *dataType* stands for the Modbus data type, *address* is the address of the target point and *value* is the value that is intended to be written into the target point.

7.1.5 GeneralDef.EventElement

EventElement:

```
public struct EventElement
{
    public GeneralDef.DataType dataType;
    public int address;
    public int value;
    public int fileNo;
}
```

Depending on the event criteria that client programs in the server, server fires events towards the client. The event data is carried in a *WriteElement* data structure. *dataType* represents the Modbus data type, *address* represents the point address, *value* represents the point value and *fileNo* represents the file number if the event is generated by a file record.

7.1.6 GeneralDef.ChannelInfoType

ChannelInfoType:

```
public struct ChannelInfoType
{
    public string serverName;
    public string serverPort;
}
```

This structure is used by advanced applications to retrieve information about the nodes connected to the Modbus server. *serverName* is the name or IP address of the connected machine and *serverPort* is the IP port that is used by the other party to connect .

7.1.7 GeneralDef.SOEDataElement

SOEDataElement:

```
public struct SOEDataElement
{
   public DateTime eventTime;
   public object eventData;
}
```

This structure is used to return the sequence of event data for function codes or data points. *eventTime* represents the time event occurred and *evenData* represents the event value. This field is different for function code and point events and is explained as follows.

7.1.8 GeneralDef.PointSOEDataElement

PointSOEDataElement:

```
public struct PointSOEDataElement
{
   public int value;
}
```

This structure carries data about a point SOE. *value* represents the point event. This structure is assigned to *eventData* of *SOEDataElement* when retrieving the SOE data.

7.1.9 GeneralDef.FCSOEDataElement

FCSOEDataElement:

```
public struct FCSOEDataElement
{
   public byte[] msgString;
}
```

This structure carries data about a function code sequence of event. *msgString* represents the exact message received on the channel. This structure is assigned to *eventData* of *SOEDataElement* when retrieving the SOE data.

7.1.10 GeneralDef.ElementGroup

ElementGroup:

```
public class ElementGroup
{
   public ArrayList list;
}
```

Client application may write data of multiple points into the server via a single call. In this case, client can include the required information as members of the *list* data member of *ElementGroup* class. This class can include *WriteElement*, *EventElement* and *EventStatusElement* data structures.

7.1.11 GeneralDef.ResultGroup

ResultGroup:

```
public class ResultGroup
{
   public ArrayList list;
}
```

Client application may write data of multiple points into the server via single call. In this case, server returns the result of operation per entry in the original request. The results are embedded in *list* data member of a *ResultGroup* class. True means that the operation for the corresponding element in the request list was successful whereas False means the operation did not go through.

7.2 IRTULink

RTULink is the object that links server and client together. RTULink exposes a set of methods or APIs that can be used by the client application to communicate with the server.

7.2.1 IRTULink.Create

bool Ctreate(int _unitID, out Guid _rtuKey);

This method is used to create an RTU in the server space.

Arguments:
_unitID: Modbus unit ID of the RTU being created.
_rtuKey: This is a return argument and is a unique number created by the server. Server identifies the RTU in subsequent calls by this number. Therefore, it is important for the client application to keep this number throughout its life cycle.

Return:
True if the operation was successful, false if not.

7.2.2 IRTULink.Enable

bool Enable(Guid _rtuKey, bool _enDis);

This method is used to enable/disable an RTU. If an RTU is not enabled, server does not respond to the Modbus messages addressed for the RTU.

Arguments:
_rtuKey: RTU key ID returned by the *Create* call.
_enDis: True means the caller wants to enable the RTU; False means the caller needs to disable the RTU

Return:
True if the operation was successful, false if not.

7.2.3 IRTULink.Destroy

bool Destroy(Guid _rtuKey);

This method is used to destroy an RTU in the server space. If client needs to use the RTU again, it needs to create it one more time

Arguments:

_rtuKey: RTU key ID returned by the *Create* call.

Return:
True if the operation was successful, false if not.

7.2.4 IRTULink.RTUExists

bool RTUExist(Guid _rtuKey);

This method is used to verify if an RTU already exists in the server database.

Arguments:
_rtuKey: RTU key ID returned by the *Create* call.

Return:
True if exists, false if not.

7.2.5 IRTULink.PopEvent

bool PopEvent(Guid _rtuKey, out GeneralDef.DataType _type, out int _address, out int _value, TimeSpan _timeout);

This method is used to receive queued events for the RTU. Server queues the events for the clients and clients need to call this method to retrieve the queued events.

Arguments:
_rtuKey: RTU key ID returned by the *Create* call.
_type: Point data type.
_address: Point address.
_value: Point value.
_timeOut: Amount of time that the call needs to wait at the server to receive events. Please note that if there is an event already in the queue waiting for the client, method call returns immediately.

Return:
True if the operation was successful, false if not.

7.2.6 IRTULink.ReadData

bool ReadData(Guid _rtuKey, GeneralDef.DataType _dataType, int _address, out int _value);

This method is used to read data from the server database.

Arguments:
_rtuKey: RTU key ID returned by the *Create* call.

_datatype: Point data type.
_address: Point address.
_value: Point value.

Return:
True if the operation was successful, false if not.

7.2.7 IRTULink.WriteData

bool WriteData(Guid _rtuKey, GeneralDef.DataType _dataType, int _address, int _value);

This method is used to write data into the server database.

Arguments:
_rtuKey: RTU key ID returned by the *Create* call.
_datatype: Point data type.
_address: Point address.
_value: Point value.

Return:
True if the operation was successful, false if not.

7.2.8 IRTULink.WriteGroupData

bool WriteGroupData(Guid _rtuKey, GeneralDef.ElementGroup _dataGroup, out GeneralDef.ResultGroup _result);

This method is used to write a group of data into the server database.

Arguments:
_rtuKey: RTU key ID returned by the *Create* call.
_dataGroup: A list containing the points' information.
_result: A list containing the result of operation per point.

Return:
True if the operation was successful, false if not.

7.2.9 IRTULink.SetEventStatus

bool SetEventStatus(Guid _rtuKey, GeneralDef.DataType _dataType, int _address, GeneralDef.EventType _eventStatus);

This method is used to set event status for a point.

Arguments:

_rtuKey: RTU key ID returned by the *Create* call.
_dataType: Point data type .
_address: Point address.
_eventStatus: Event registration definition for the point.

Return:
True if the operation was successful, false if not.

7.2.10 IRTULink.SetGroupEventStatus

```
bool SetGroupEventStatus(Guid _rtuKey, GeneralDef.ElementGroup _eventStatusGroup,
 out GeneralDef.ResultGroup _result);
```

This method is used to set event status for a group of points.

Arguments:
_rtuKey: RTU key ID returned by the *Create* call.
_eventStatusGroup: List of points information and the requested event registration.
_result: List containing the corresponding results of the operation.

Return:
True if the operation was successful, false if not.

7.2.11 IRTULink.SetDatabaseSize

```
bool SetDatabaseSize(Guid _rtuKey, GeneralDef.DataType _dataType, int _size);
```

This method is used to specify the database size for each type of the points. The database size can be set up to 65535.

Arguments:
_rtuKey: RTU key ID returned by the *Create* call.
_dataType: Point data type
_size: Desired database size.

Return:
True if the operation was successful, false if not.

7.3 IAdvAppLink

AdvAppLink is the object that allows the client to access advanced application information.

7.3.1 IAdvAppLink.ReadPointChangeNo

bool ReadPointChangeNo(int _unitID, GeneralDef.DataType _type, int _address, out uint _changeNo, out bool _rollOver, out DateTime _startTime);

This method is used to read number of changes per point from the start up or previous reset.

Arguments:
_unitID: RTU unit ID.
_type: Point data type
_address: Point address.
_changeNo: Number of changes in value.
_rollOver: If true, indicates that the counter has rolled over. The overflow value is the maximum value for unsigned integer.
_startTime: The time that logging of changes started.

Return:
True if the operation was successful, false if not.

7.3.2 IAdvAppLink.ClearPointChangeNo

bool ClearPointChangeNo(int _unitID, GeneralDef.DataType _type, int _address);

This method is used to clear the number of changes for a point.

Arguments:
_unitID: RTU unit ID.
_type: Point data type
_address: Point address.

Return:
True if the operation was successful, false if not.

7.3.3 IAdvAppLink.ClearPointChangeNobyType

bool ClearPointChangeNobyType(int _unitID, GeneralDef.DataType _type);

This method is used to clear the number of changes for all the points with a specific type.

Arguments:
_unitID: RTU unit ID.
_type: Point data type

Return:
True if the operation was successful, false if not.

7.3.4 IAdvAppLink.ClearPointChangeNoAll

bool ClearPointChangeNoAll(int _unitID);

This method is used to clear the number of changes for all the points of a specific RTU.

Arguments:
_unitID: RTU unit ID.

Return:
True if the operation was successful, false if not.

7.3.5 IAdvAppLink.ClearPointChangeNoGlobal

bool ClearPointChangeNoGlobal();

This method is used to clear the number of changes for all the points of all the RTUs.

Arguments:

Return:
True if the operation was successful, false if not.

7.3.6 IAdvAppLink.ReadFCMasNo

bool ReadFCMsgNo(int _unitID, int _fc, out uint _msgNo, out bool _rollOver, out DateTime _startTime);

This method is used to get the number of received messages per function code.

Arguments:
_unitID: RTU unit ID.
_fc: Function code.
_msgNo: Number of messages.
_rollOver: If true, indicates that the counter has rolled over. The overflow value is the maximum value for unsigned integer.
_startTime: The time that logging of changes started.

Return:
True if the operation was successful, false if not.

7.3.7 IAdvAppLink.ClearFCMasNo

bool ClearFCMsgNo(int _unitID, int _fc);

This method is used to clear the number of received messages per function code.

Arguments:
_unitID: RTU unit ID.
_fc: Function code.

Return:
True if the operation was successful, false if not.

7.3.8 IAdvAppLink.ClearFCMasNoAll

bool ClearFCMsgNoAll(int _unitID);

This method is used to clear the number of received messages for all function codes of an RTU.

Arguments:
_unitID: RTU unit ID.

Return:
True if the operation was successful, false if not.

7.3.9 IAdvAppLink.ClearFCMasNoGlobal

bool ClearFCMsgNoGlobal();

This method is used to clear the number of received messages for all function codes of all RTUs.
Arguments:

Return:
True if the operation was successful, false if not.

7.3.10 IAdvAppLink.BlockPoint

bool BlockPoint(int _unitID, GeneralDef.DataType _type, int _address, bool _blockStatus);

This method is used to block a point so that value of the point cannot be changed. This is used as a safety feature to block execution of some controls temporarily when for example the field crew is working on the equipments.

Arguments:
_unitID: RTU unit ID.
_type: Point data type.
_address: Point address.
_blocktatus: Blocking status, if set to true point is blocked. If the to false, point is unblocked.

Return:
True if the operation was successful, false if not.

7.3.11 IAdvAppLink.ReadBlockStatus

bool ReadBlockStatus(int _unitID, GeneralDef.DataType _type, int _address, out bool _blockStatus, out DateTime _blockTime);

This method is used to read blocking status of a point.

Arguments:
_unitID: RTU unit ID.
_type: Point data type.
_address: Point address.
_blocktatus: Blocking status, if set to true point is blocked. If the to false, point is unblocked.

Return:
True if the operation was successful, false if not.

7.3.12 IAdvAppLink.BrowseChannels

bool BrowseChannels(out ArrayList _channelsList);

This method is used to browse the IP information of the Modbus clients connected to the server.

Arguments:
_channelList: Array list containing channel information. Array members of type *ChannelInfoType*.

Return:
True if the operation was successful, false if not.

7.3.13 IAdvAppLink.ReadPointSOE

bool ReadPointSOE(int _unitID, GeneralDef.DataType _type, int _address, out ArrayList _pointSOE);

This method is used to retrieve sequence of events for a point. After reading the SOE, the SOE table at the server is cleared. Please note that the first entry in the list represents the oldest event while the last entry represents the newest one.

Arguments:
_unitID: RTU unit ID.
_type: Point data type.
_address: Point address.

_pointSOE: Array list containing SOE information. The array members are of type *SOEDataElement*. The *value* field is the *SOEDataElement* is set to an object of type *PointSOEDataElement*.

Return:
True if the operation was successful, false if not.

7.3.14 IAdvAppLink.ReadFCSOE

bool ReadFCSOE(int _unitID, int _fc, out ArrayList _fcSOE);

This method is used to retrieve sequence of events for a function code. After reading the SOE, the SOE table at the server is cleared. Please note that the first entry in the list represents the oldest event while the last entry represents the newest one.

Arguments:
_unitID: RTU unit ID.
_fc: Function code.
_fcSOE: Array list containing SOE information. The array members are of type *SOEDataElement*. The *value* field is the *SOEDataElement* is set to an object of type FC*SOEDataElement*.

Return:
True if the operation was successful, false if not.

8. NetProModServLiteCE APIs

NetProModServLiteCE is the class that implements Modbus communication.

8.1 NetProModServLiteCE. NetProModServ

public NetProModServLiteCE(int _modbusTCPPort, int _noofRTUs, bool _advancedApplications,
string _clientID, string _license, out bool _licenseValid)

This is the class constructor.

Arguments:
_modbusTCPPort: IP port number that is supposed to listen to incoming Modbus calls.
_noofRTUs: No of RTUs that the code supports according to the license.
_advancedApplications: If true, it indicates that the code is supposed to support advanced applications. Pleas note that you may need to purchase or upgrade proper license for this feature.
_clientID: The client ID that is generated through the licensing process.
_license: License code.
_licenseValid: If true indicates that the code has accepted the license along with the input parameters.

Return:

8.2 NetProModServLiteCE. Process

bool Process();

This method is used to process the incoming Modbus traffic. The method needs to be called periodically.

Arguments:

Return:
True if the operation was successful, false if not. If the license was not accepted by the class at the initialization, the method returns with false for each call after 30 minutes of initializations.

8.3 NetProModServLiteCE. Shutdown

public void ShutDown();
This method is used to shutdown the class to release all the resources.

Arguments:

Return:

8.4 NetProModServLiteCE. CreateRTULink

public IRTULink CreateRTULink()

This method is used to create *RTULink* object in the server. Please note that you only need to create this object once even if you are intending to support more than one RTU.

Arguments:

Return:
RTU link object pointer.

8.5 NetProModServLiteCE. GetAdvAppLink

public IAdvAppLink GetAdvAppLink()

This method is used to acquire a pointer to the *AdvAppLink* object in the server.

Arguments:

Return:
Advanced application object pointer.