TRIANGLE MICROWORKS, INC.

.NET Protocol Components
User Manual

Version 3.13.000

November 8, 2011

Property of Triangle MicroWorks, Inc.

Chapter 1 Introduction

Congratulations on your purchase of a Triangle MicroWorks, Inc. (TMW) .NET Communications Protocol Component (.NET.SCL). This product is a member of a family of communication protocol libraries supported by Triangle MicroWorks, Inc. This library was designed to be flexible, easy to use, and is available for the development of communications applications on the Microsoft .NET framework. For these reasons, the Triangle MicroWorks, Inc. .NET Communications Protocol Components will provide an excellent solution to your communication protocol requirements.

This document is the user manual for all of the TMW .NET Communications Protocol Components. The manual begins with generic information pertinent to all of the TMW .NET Communications Protocol Components and gets more specific in later chapters. In addition, a help file documenting the classes available in the components and a Configuration and Interoperability (C-I) guide is included for each purchased protocol.

This manual also contains sections specific to the Source Code versions of the TMW.NET Communications Protocol Components. The source code versions are available for purchase for each of the Protocol Components.

Please Note

Your license agreement limits the installation of this component to specific products. Before installing this component in a new target application, check your license agreement to verify that it allows use on that target application. Contact Triangle MicroWorks, Inc. for information about extending the number of target applications that may use this component.

1.1 Supported Protocols

As mentioned above, the TMW .NET Protocol Components support a number of different protocols. The list of communication protocols currently supported can be found in Table 1. This table lists the protocol name, the protocol family, and the abbreviation used both within this manual and the throughout the associated TMW .NET Protocol Components for each protocol.

Table 1: Supported Protocols

Protocol Name	Protocol Family	Abbreviation(s)/Prefix(es)
IEC 60870-5-101	i870	m101, s101, ft12
IEC 60870-5-102	i870	m102, s102, ft12
IEC 60870-5-103	i870	m103, s103, ft12
IEC 60870-5-104	i870	m104, s104, i104

DNP3	dnp	mdnp, sdnp, dnp
MODBUS	modbus	mmb, smb, mb

Each of the protocols listed above is available as a Master (Controlling Station), or Slave (Controlled Station) implementation. As shown in **Figure 1**, a typical substation topology consists of an off-site central station computer that polls one or more Remote Terminal Units (RTU) in the substation. In turn, each Master polls one or more Intelligent Electronic Devices (IED) that monitors and controls the substation. **Figure 1** illustrates how Master and Slave Components might be used in a typical substation.

Figure 1: Typical Substation Communication Topology

All of the TMW .NET Protocol Components share a common architecture, allowing them to be installed in any combination required. For example, it is straightforward to implement a device that supports DNP3 and IEC 60870-5-101. Combining protocols is discussed in detail in Section 5.1.

1.2 Manual Overview

This section describes the layout of this manual and is useful in determining which sections of the manual are pertinent to a particular situation. As mentioned above, all of the TMW .NET Protocol Components share a common architecture. This manual documents all of the supported TMW .NET Protocol Components. The manual begins with generic information that applies to all of the TMW .NET Protocol Components. It then provides more specific information.

Chapter 1 provides an introduction to TMW .NET Protocol Components and describes the layout of this manual and conventions used throughout this document.

Chapter 2 presents a Quick Start Guide that is intended to provide enough information to get a user with communication protocol experience and/or experience with a previous TMW .NET Protocol Component "up and running" quickly. It also provides a framework that less experienced users can use as a guide while progressing through the more detailed sections of the manual.

Chapter 3 provides an overview of the TMW .NET Protocol Components. Separate subsections describe Master-, Slave-, and Peer-specific details.

Chapter 4 describes the specifics related to the Source Code version of TMW .NET Protocol Components. Separate subsections describe implementation issues for Master, Slave, and Peer libraries.

Chapter 5 provides information on advanced topics. These topics are often library dependent.

Chapter 6 provides guidelines and suggestions on debugging and testing your implementation. These guidelines are techniques that have proven useful in previous porting and development efforts.

1.3 Manual Conventions and Abbreviations

The following conventions and abbreviations are used throughout this manual.

TMW – Triangle MicroWorks, Inc.

<*scl*> - Replace with appropriate TMW .NET Protocol Components prefix from Table 1.

<*family*> - Replace with appropriate TMW .NET Protocol Components family from Table 1.

xxx – This abbreviation is used to represent a family of functions with similar names (e.g., a family of functions that exist for each DNP object or IEC 60870-5 ASDU type. Replace xxx with the appropriate object or type.

Chapter 2 Quick Start Guide

2.1 Quick Start Guide for TMW .NET Protocol Components

This will provide users with previous experience with communication protocols, and/or the TMW architecture, a quick step-by-step guide to integrating the TMW .NET Protocol Components on their platform. First time users will also get a feel for the steps required to implement a TMW .NET Protocol Components on their platform.

This will serve to highlight each step of the implementation process and provide a very high level description of what is required. If more information is required, please refer to the sample applications and the detailed sections found later in this manual. For the sake of brevity not every situation can be explained in detail. Again, if there are any questions as to whether a specific step is appropriate to your TMW .NET Protocol Components please refer to the detailed sections later in the manual.

2.2 Install the TMW .NET Protocol Components

The TMW .NET Protocol Components are delivered as a standard windows setup file. For information about the Source Code Library version see Chapter 3.

The TMW .NET Protocol Components files should be installed in the desired target directory (<*installdir*>). TMW recommends retaining the directory structure of the SCL as it is supplied. To install TMW .NET Protocol Components simply run the setup file and follow the prompts.

We have included Visual Studio 2008 integrated help for the TMW .Net Protocol Components. To install this make sure Visual Studio is shutdown and run the installation setup program 'Install VS 2008 Help' from the Start Menu\Program Files\Triangle MicroWorks\.NET Protocol Components\Help. After this is installed selecting a object exported by the .NET Protocol Components such as MDNPRequest or M14Request in Visual studio and select F1 to get detailed help on the requests available.

In addition to VS 2008 integrated help we have also included standard compiled html help. This help can be accessed from the Start Menu\Program Files\Triangle MicroWorks\.NET Protocol Components\Help menu. All of the classes and methods exported by the TMW .NET Components may be viewed from this help document.

We have also included some sample programs to get you started. The samples that contain a Graphical User Interface (GUI) can be accessed at: Start Menu\Program Files\Triangle MicroWorks\.NET Protocol Components\Samples. These files should be used as a starting point for writing your own applications. The source code for all of the samples is located at: <installdir>\Samples.

The default <installdir> would normally be C:\Program Files\Triangle MicroWorks\.NET Protocol Components.

2.2.1 Included Samples

2.2.1.1 DNP

DNPMasterGUI – a Windows Forms application that implements a simple DNP master .

DNPMasterGUI.VBNet – a Windows Forms Visual Basic application that implements a simple DNP master.

DNPSlaveGUI – a Windows Forms application that implements a simple DNP slave (outstation).

DNPSlaveGUI.VBNet – a Windows Forms Visual Basic application that implements a simple DNP slave (outstation).

DNPMasterModem – a Windows Forms application that implements a simple DNP master that opens a channel disabled and allows modem control before the DNP protocol starts communicating.

DNPMasterFileTransfer – a Windows Forms application that implements a simple DNP master that allows you to open/close/read and write files and read directories. It uses the built in database interface to the Windows File System while using database events for other data types.

DNPMasterDatasets – a Windows Forms application that implements a DNP master that provides an interface for data set configuration, reads and writes.

DNPSlaveDatasets – a Windows Forms application that implements a DNP slave (outstation) that provides an interface for data sets.

DNPDatabaseEvents – shows how to use the event based model for a DNP master and slave (outstation).

DNPDatabaseEvents.VBNet – shows how to use the event based model for a DNP master and slave (outstation) in Visual Basic.

DNPMasterDatabaseEvents – shows how to use the event based model for a DNP master .

DNPSlaveDatabaseEvents – shows how to use the event based model for a DNP slave (outstation).

2.2.1.2 I101

I101MasterGUI – a Windows Forms application that implements a simple IEC 60870-5-101 master

I101SlaveGUI – a Windows Forms application that implements a simple IEC 60870-5-101 slave

I101MasterDatabaseEvents – shows how to use the event based model for an IEC 60870-5-101 master.

I101SlaveDatabaseEvents – shows how to use the event based model for an IEC 60870-5-101 slave.

2.2.1.3 I102

I102SlaveGUI – a Windows Forms application that implements a simple IEC 60870-5-102 slave

I102MasterGUI – a Windows Forms application that implements a simple IEC 60870-5-102 master

I102DatabaseEvents – shows how to use the event based model for an IEC 60870-5-102 master and slave

I102MasterDatabaseEvents – shows how to use the event based model for an IEC 60870-5-102 master.

I102SlaveDatabaseEvents – shows how to use the event based model for an IEC 60870-5-102 slave.

I103
I103SlaveGUI – a Windows Forms application that implements a simple IEC 60870-5-103 slave.

I103MasterGUI – a Windows Forms application that implements a simple IEC 60870-5-103 master.

I103DatabaseEvents – shows how to use the event based model for an IEC 60870-5-103 master and slave.

I103MasterDatabaseEvents – shows how to use the event based model for an IEC 60870-5-103 master.

I103SlaveDatabaseEvents – shows how to use the event based model for an IEC 60870-5-103 slave.

I104

I104MasterGUI – a Windows Forms application that implements a simple IEC 60870-5-104 master

I1104SlaveGUI – a Windows Forms application that implements a simple IEC 60870-5-104 slave

I104RedundantMasterGUI – a Windows Forms application that implements a simple IEC 60870-5-104 master with redundancy enabled.

I104RedundantSlaveGUI – a Windows Forms application that implements a simple IEC 60870-5-104 slave with redundancy enabled.

I104MasterDatabaseEvents – shows how to use the event based model for an IEC 60870-5-104 master.

I104SlaveDatabaseEvents – shows how to use the event based model for an IEC 60870-5-104 slave.

2.2.1.4 Modbus

MMBWebService – a web service application that wraps a Modbus master as a web service.

SMBSimulator – a Windows Forms application that implements a simple Modbus slave that can be used with the MMBWebService sample.

MBDatabaseEvents – shows how to use the event based model for a Modbus master and slave.

MBMasterDatabaseEvents – shows how to use the event based model for a Modbus master.

MBSlaveDatabaseEvents – shows how to use the event based model for a Modbus slave

2.3 Create Interoperability Guide

Your documentation should include an Interoperability Document, as described in the IEC 60870-5 and DNP standards.

The TMW .NET Protocol Components includes an example Configuration/Interoperability Guide. This guide is named <*scl*> *CI Guide.rtf* and is located in the <*installdir* > directory. This document is a template with much of the information already supplied, indicating the features of the Source Code Library. You will need to edit the document to indicate the features your device will support and to describe methods of configuration.

2.4 Build the TMW .NET Protocol Components and Samples

The first step in implementing the TMW .NET Protocol Components should be to build the samples before making any modifications. Once installed, all TMW .NET Protocol Components sample code and/or binary files reside in the subdirectories under <*installdir*>.

For the source code version of TMW .NET Protocol Components all of the source files in each of these subdirectories must be compiled. This step confirms proper installation of the components as well as highlighting any compiler-specific issues. For information about the Source Code directory structure and other details refer to Chapter 3.

The project and source files for the .NET Component sample applications reside in <*installdir*>*Samples*\.... To build a sample application, open the supplied Visual Studio project file (xxx.csproj) and build it with Visual Studio. This will create a solution file which may then be saved.

You should refer to these sample applications when completing the steps in the Quick Start Guide and building your own application.

2.5 Common TMW .NET Protocol Components Setup

Calls into the TMW .NET Protocol Components are limited to the following areas: licensing; application initialization; database initialization and managing channels, sessions, and (for IEC 60870-5 protocols) sectors.

Licensing, application, and database initialization is described in the following sections. Managing channels, sessions, and sectors is described in Section 2.7.

2.5.1 Licensing

When the TMW .NET Protocol Components are first installed on your machine they come with an initial Trial License. This license provides full functionality for a period of 21 days to evaluate the Protocol Components for purchase. During this trial period an application will only be allowed to run for 1 hour.

Triangle MicroWorks provides three TMW .NET Protocol Components Licensing options beyond the free trial license.

1. Single Use License – This is a license that is tied to a specific computer.
2. Redistributable License – This form gives the developer the right to re-distribute the TMW .NET Protocol Components with a single application to run on an unlimited number of computers.
3. Redistributable with Source Code License – This is similar to the Redistributable License except that Source code is provided to the developer.

2.5.1.1 Single Use License

Single use licensing supports installation and use on a single machine. The license is specific for the machine the software is installed on. In order to use single use licensing, the licensing tool is used to request an activation key. Once the activation key is received

from Triangle MicroWorks Inc. it is then installed on the target machine using this same tool. The tool is accessible from the Start menu under the Licensing item.

No licensing changes need to be made to the startup code in the example files or in your own application. The software will detect the single use license and functionality based on that license will be allowed.

2.5.1.2 Redistributable License

When using this form of TMW .NET Protocol Components product licensing, the customer will receive a unique license key from Triangle MicroWorks Inc. upon payment and execution of the license agreement. This key should be included in the product the customer is creating as follows:

During initialization of the application (usually in the constructor) register for the TMWApplication.ObtainLicenseEvent. Note that the event handler should be established prior to the TMWApplication constructor which is called by
'`new TMWApplicationBuilder()`'

```
TMWApplication.ObtainLicenseEvent += new
TMWApplication.ObtainLicenseEventDelegate(TMWApplication_ObtainLicenseE
vent);
```

And implement the following event handler:

```
TMW.SCL.NETLicense TMWApplication_ObtainLicenseEvent()
{
  NETLicense license = new NETLicense();
  license.setLicenseKey("KeyFromTMW");
  return license;
}
```

The text "`KeyFromTMW`" in the event handler should be replaced with the actual license key provided by Triangle MicroWorks Inc.

2.5.1.3 Source Code Version

No licensing changes need to be made to the startup code in the example files or in your own application. The source code version does not require a license to run.

2.5.1.4 Licensing Error Message

Licensing errors by default will pop up Windows message boxes indicating the problem. When running as a Windows Service you will want to disable these message boxes. Whether disabled or not you can register to receive License error messages and write them to the Windows Event Log or otherwise record them. For a Trial or Single Use License this should be done before the TMWApplication is constructed, while for a Redistributable License this should be done before the NETLicense::setLicenseKey is called.

For the Single Use or Trial License

```
  TMWApplication.DisableMessages = true;
  TMWApplication.ErrorMessageEvent +=
    new TMWApplication.ErrorMessageEventDelegate(
        TMWApplication_ErrorMessageEvent);
  TMWApplicationBuilder applBuilder =
    new TMWApplicationBuilder();
  pAppl = TMWApplicationBuilder.getAppl();
```

or for the Redistributable License

```
TMW.SCL.NETLicense TMWApplication_ObtainLicenseEvent()
 {
  NETLicense license = new NETLicense();
  license.DisableMessages = true;
  license.ErrorMessageEvent +=
      new NETLicense.ErrorMessageEventDelegate(
          TMWApplication_ErrorMessageEvent);
  license.setLicenseKey("KeyFromTMW");
  return license;
}
      return license;
    }
```

2.5.2 Application Initialization

All protocols share a single application object: TMWApplication. This object is created by creating a TMWApplicationBuilder object and asking for the TMWApplication object using the getAppl() method. The TMWApplication object has an application initialization method, InitEventProcessor(), which must be called once to initialize the TMW .NET Protocol Components. It also has a property, EnableEventProcessor, that should be set to true to periodically check if data is available on any of the communication channels.

```
TMWApplicationBuilder applBuilder = new TMWApplicationBuilder();
TMWApplication pAppl = TMWApplicationBuilder.getAppl();
pAppl.InitEventProcessor();
pAppl.EnableEventProcessor = true;
```

The property TMWApplication.LoopPeriod is used to specify the rate in milliseconds at which checks for data on a channel are performed.

2.5.3 Database Initialization

By default, all of the TMW .NET Protocol Components use a built-in database to store protocol data. Should you need to override this behavior, set the TMWSimDataBase.UseSimDatabase property to false. This will enable the event based database model allowing use of a custom user defined database. Most slave components s support a UseSimControlDatabase property also. This allows a slave application to

separate support for Output Controls from the other database functionality. Examples of this can be found in the xxxDatabaseEvents samples.

2.6 TMW .NET Protocol Components Target

The TMW .NET Protocol Components uses the WinIoTarg.dll to implement the target layer communications for TCP/IP, RS232, etc… No user implementation of a target layer is required.

2.7 Manage Channels, Sessions, and Sectors

Channels, sessions, and sectors (in some protocols) are managed by methods, properties, and events found in the TMWChannel, TMWSession, and TMWSector classes and their protocol specific derivations.

These classes provide methods and events for opening, modifying, and closing channels, sessions, or sectors. Each of these classes provides configuration properties that can be used to configure the object. The constructors of these objects initialize the objects to common default values. After creating an object, the target application should modify the configuration properties to any specific values required by your application prior to opening the Channel, Session or Sector. To see all of the properties available for a .NET Component class you should use the integrated or html help functionality described above. For example select DNPChannel and enter F1 or search for DNPChannel in the html help file accessible from the Start Menu.

2.8 Install the TMW .NET Protocol Components

Here is a typical example (open a DNP Master) of this initialization process:

```
public partial class MasterForm : Form
{
  private DNPChannel masterChan;
  private MDNPSession masterSesn;

 …

  public MasterForm()
  {
    ...
    masterChan = new DNPChannel(TMW_CHANNEL_OR_SESSION_TYPE.MASTER);

    masterChan.Type = WINIO_TYPE.TCP;
    masterChan.Name = ".NET DNP Master";  // name displayed in analyzer window
    masterChan.WinTCPipAddress = "127.0.0.1";
    masterChan.WinTCPipPort = 20000;
    masterChan.WinTCPmode = TCP_MODE.CLIENT;
    masterChan.OpenChannel();

    masterSesn = new MDNPSession(masterChan);
    masterSesn.AuthenticationEnabled = false;

    masterSesn.OpenSession();
    ...
```

After opening the session it should be possible to test communications with a remote device. Note that the TMWApplication initialization code is not shown in this code segment, but should be performed as shown above.

2.9 The TMW .NET Protocol Components Database

The TMW .NET Protocol Components database is the mechanism for dealing with the processing of the protocol specific values, flags, timestamps, qualities, etc. in your application.

The interface between the TMW .NET Protocol Components and the application database is a set of events defined in TMWSimDatabase and its protocol specific derivations or a set of predefined TMWSimPoints and its protocol specific derivations.

There are 2 ways to use the database classes.

1. Use the built in simple database.
2. Use the .NET event interface in the database.

When the TMW .NET Protocol Components is shipped, the simple "simulated" database is enabled. In this mode the protocol data (i.e. values, flags, timestamps, qualities, etc.) are managed by the built in database and provided through the TMWSimDatabase and TMWSimPoint class hierarchy. After opening, each protocol specific Session or Sector will have a database associated with it. This database may be accessed through the Session or Sectors SimDatabase property. The SimDatabase has an UpdateDBEvent event associated with it that provides an event to notify the application of TMWSimPoint changes.

Here is an example using the built in database:

```
private void UpdateDBEvent(TMWSimPoint simPoint)
{
  if (InvokeRequired)
    Invoke(new UpdatePointDelegate(UpdateDBEvent), new object[] { simPoint });
  else
  {
    switch (simPoint.PointType)
    {
      case 1:
        // Binary Input
        updateBinaryInput(simPoint);
        break;
      case 10:
        // Binary Output (CROB)
        updateBinaryOutput(simPoint);
        break;
      case 20:
        // Binary Counters
        updateBinCntr(simPoint);
        break;
      case 30:
        // Analog Inputs
        updateAnalogInput(simPoint);
        break;
      case 40:
        // Analog Output
        updateAnalogOutput(simPoint);
```

```
        break;
      default:
        protocolBuffer.Insert("Unknown point type in database update routine");
        break;
    }
   }
}

...

db = (MDNPSimDatabase)masterSesn.SimDatabase;
// Register to receive notification of database changes
db.UpdateDBEvent += new TMWSimDataBase.UpdateDBEventDelegate(UpdateDBEvent);
```

The 2nd database model disables the built in database and utilizes .NET events on the TMWSimDatabase and its derivations to notify the application of various protocol events. This mechanism provides a tighter integration to the protocol and leaves the management of protocol data to the application developer. This model is more appropriate when developing an application that provides its own database.

```
private static void OpenSlave()
{
  slaveChan = new DNPChannel(TMW_CHANNEL_OR_SESSION_TYPE.SLAVE);

  slaveChan.Type = WINIO_TYPE.TCP;
  slaveChan.Name = ".NET DNP Slave";  /* name displayed in analyzer window */
  slaveChan.WinTCPipAddress = "127.0.0.1";
  slaveChan.WinTCPipPort = 20000;
  slaveChan.WinTCPmode = TCP_MODE.SERVER;
  slaveChan.OpenChannel();

  slaveSesn = new SDNPSession(slaveChan);
  slaveSesn.UnsolAllowed = true;

  slaveSesn.OpenSession();

  sdb = (SDNPDatabase)slaveSesn.SimDatabase;

  SDNPDatabase.UseSimDatabase = false;
  SDNPDatabase.UseSimControlDatabase = false;


  // binary inputs
  sdb.BinInQuantityEvent += new
SDNPDatabase.BinInQuantityDelegate(SlaveBinInQuantityEvent);
  sdb.BinInGetPointEvent += new
SDNPDatabase.BinInGetPointDelegate(SlaveBinInGetPointEvent);
  SDNPDatabase.BinInReadEvent += new SDNPDatabase.BinInReadDelegate(SlaveBinInReadEvent);

  // analog inputs
  sdb.AnlgInQuantityEvent += new
SDNPDatabase.AnlgInQuantityDelegate(SlaveAnlgInQuantityEvent);
  sdb.AnlgInGetPointEvent += new
SDNPDatabase.AnlgInGetPointDelegate(SlaveAnlgInGetPointEvent);
  SDNPDatabase.AnlgInReadEvent += new
SDNPDatabase.AnlgInReadDelegate(SlaveAnlgInReadEvent);

// binary outputs
  sdb.BinOutQuantityEvent += new
SDNPDatabase.BinOutQuantityDelegate(SlaveBinOutQuantityEvent);
  sdb.BinOutGetPointEvent += new
SDNPDatabase.BinOutGetPointDelegate(SlaveBinOutGetPointEvent);
```

```
  SDNPDatabase.BinOutReadEvent += new
SDNPDatabase.BinOutReadDelegate(SlaveBinOutReadEvent);
  SDNPDatabase.BinOutSelectEvent += new
SDNPDatabase.BinOutSelectDelegate(SlaveBinOutSelectEvent);
  SDNPDatabase.BinOutOperateEvent += new
SDNPDatabase.BinOutOperateDelegate(SlaveBinOutOperateEvent);

  // analog outputs
  sdb.AnlgOutQuantityEvent += new
SDNPDatabase.AnlgOutQuantityDelegate(SlaveAnlgOutQuantityEvent);
  sdb.AnlgOutGetPointEvent += new
SDNPDatabase.AnlgOutGetPointDelegate(SlaveAnlgOutGetPointEvent);
  SDNPDatabase.AnlgOutReadEvent += new
SDNPDatabase.AnlgOutReadDelegate(SlaveAnlgOutReadEvent);
  SDNPDatabase.AnlgOutSelectEvent += new
SDNPDatabase.AnlgOutSelectDelegate(SlaveAnlgOutSelectEvent);
  SDNPDatabase.AnlgOutOperateEvent += new
SDNPDatabase.AnlgOutOperateDelegate(SlaveAnlgOutOperateEvent);

}
```

The UseSimDatabase and UseSimControlDatabase properties on the TMWSimDatabase switches between these two modes. Note that in the above example they are set to false.

Note also that the event handlers are registered with the database so that your application can process the requests through the protocol.

2.10 Issue Commands (Master Sessions)

For Master Sessions, commands are issued by calling methods on the TMWRequest class and its derivations. The request can have an event specified that will notify the application of the request status (i.e. completion, failure, etc…).

When the request is completed, the TMW .NET Protocol Components will call the event. The event arguments include the request and the received response.

Information on protocol-specific commands is given in the protocol-specific integrated or html help described above.

Here is an example of issuing a request (Master DNP Binary output command):

```
private void BinOutOn_Click(object sender, EventArgs e)
{
  // Determine which point this request is for
  ushort pointNumber = Convert.ToUInt16((sender as Control).Tag);

  // Build and send the request
  CROBInfo crobData = new CROBInfo(pointNumber, CROB_CTRL.LOFF, 1, 0, 0);
  CROBInfo[] crobArray = { crobData };

  MDNPSimBinOut point = db.LookupBinOut(pointNumber);
  MDNPRequest request = new MDNPRequest(masterSesn);
  request.BinaryCommand(MDNPRequest.DNP_FUNCTION_CODE.SELECT, true, true, 100,
(byte)MDNPRequest.DNP_QUALIFIER.Q_8BIT_INDEX, crobArray);
}
```

2.11 Add Support for Change Events (Slave Sessions)

For Slave sessions, the TMW .NET Protocol Components can be configured to periodically scan for change events by setting the appropriate property on the Session or Sector. Alternately, events can be generated directly by calling the appropriate AddEvent method on the TMWSimPoint object derivation. To have more control over the event data or if the simple built-in database is not being used methods on the session or sector can also be used to add events, for example SDNPSession::BinInAddEvent(..) or S14Sector::MspAddEvent(..);

Enabling scanning for change events is a configuration option that is specified when opening the session or sector.

Examples of this can be found in the slave sample applications.

2.12 Test your Implementation

To test your implementation, you will need a system or application to act as the 'opposite' end of the communication link. For example, if you are implementing a Slave device, you will need a corresponding Master.

Triangle MicroWorks provides a flexible, scriptable Test Harness that can be used for this purpose. The Communication Protocol Test Harness is a Windows application that acts as a simple Master or Slave device and can be programmed with an automated test sequence through a scripting capability.

The protocol analyzer output from the Test Harness allows you to verify the proper operation of the device you are testing. It also allows you to generate a reference protocol analyzer output file that can be used for comparison to later tests to verify only intended modifications have occurred.

For more information please see the Triangle MicroWorks, Inc. web site at *'http://www.TriangleMicroWorks'*.

Chapter 3 Overview

This section provides an overview of the TMW .NET Protocol Components architecture. This information will be of interest to all TMW .NET Protocol Components users. Unless otherwise noted, the information in this section pertains to all of the TMW .NET Protocol Components. Note that parts of this section only apply to the source code version of TMW .NET Protocol Components.

3.1 Coding Conventions

All TMW .NET Protocol Components follow the same coding conventions. These conventions are used to specify file, function, and variable names, as well as other details. While a detailed knowledge of TMW coding conventions is not required, a few tips will help users more efficiently understand the TMW .NET Protocol Components.

Most of the files that ship with the TMW .NET Protocol Components are for internal use. See the Source Code Library documentation and the TMW .NET Protocol Components help files for details.

3.2 Architecture

This section describes the overall architecture of a TMW .NET Protocol Components. All of the TMW .NET Protocol Components share a common architecture and leverage common code. They are built on top of the Source Code Library. Not all of the information in this section is required to successfully integrate a TMW .NET Protocol Components. However, the information in this section is useful in understanding how the component behaves.

3.2.1 Layers

The TMW .NET Protocol Components architecture is broken into four layers: Physical, Link, Transport, and Application. Each of these layers provides different functionality and is designed to work completely independent of the layers above and below it in the protocol stack. While a thorough understanding of the different layers is not essential, a basic knowledge will help in implementing, tuning, and debugging a TMW .NET Protocol Components implementation.

It is important to note that although the TMW .NET Protocol Components architecture is designed to support four layers, not all protocols require or use all of the layers.

In a typical device, the user will not interface directly with the different layers. The Physical, Link, and Transport layers are all managed as a single channel, and the Application layer is managed by Sessions and Sectors.

The typical implementation does not require direct management of the Physical, Link, and Transport layers. Typically, the configuration parameters for the Physical, Link, and Transport layers are specified as part of the channel configuration.

3.2.1.1Physical Layer

The physical layer manages the target input/output (I/O) device. It accesses the actual physical device through the appropriate routines in *WinIoTarg.dll*. The physical layer is responsible for:

1) Opening and closing the physical device
2) Transmitting and receiving data on the physical channel
3) Handling low level I/O errors when transmitting or receiving data

The physical layer manages the target input/output device by opening and closing the device as required, configuring the device, etc. The physical layer optionally manages parameters related to the transmission and reception of bytes, such as inter-character timeouts, transmission delays, etc.

3.2.1.2 Link Layer

The link layer provides low-level framing, error detection, and retries. The link layer receives frames from the transport layer, adds the required link layer header and error detection information, and sends the result to the physical layer.

As the physical layer receives bytes, it passes them to the link layer, which validates the error detection, removes the link header, and sends received frames to the transport layer.

In general, the link layer pulls data from the layer above it when the link layer is available to transmit data. The application or transport layer calls a link layer method to tell the link layer that data is available. If the link layer is able to transmit, it will immediately ask the application/transport layer for the data. If the link layer is not currently available, it will ask the application/transport layer for the data as soon as the link layer becomes available.

The link layer receives bytes from the physical layer through a parsing routine. It then assembles frames as required by the protocol and passes the frames on to the transport or application layer.

3.2.1.3 Transport Layer

The transport layer, if used, breaks application layer messages into link layer frames. It also rebuilds application layer messages as they are received. The transport layer includes the sequencing required to guarantee proper transmission and reception of an entire application layer fragment.

As with the link layer, the transport layer pulls data from the layer above when the transport layer is free to transmit data.

The transport layer assembles received frames into application layer fragments and calls the application layer parsing routine to process the fragments.

3.2.1.4 Application Layer

The application layer provides application specific functionality and generates application messages to transmit. It also processes received application messages. This layer is responsible for generating and processing protocol specific requests.

3.2.2 Data Model

Each TMW .NET Protocol Components is managed through a set of hierarchical classes called channels, sessions, sectors, and points. Figure 2 below illustrates how each of these objects map to a typical application. In addition to managing the TMW .NET Protocol Components, these objects are used to access specific data points in the system.

In Figure 2, the master station can communicate using multiple communication channels (shown as COM1 and COM2 below), to multiple remote slave devices. Each remote slave device may subdivide its data into sectors, each of which may contain multiple data points. This model and these terms are further described in the sections below.

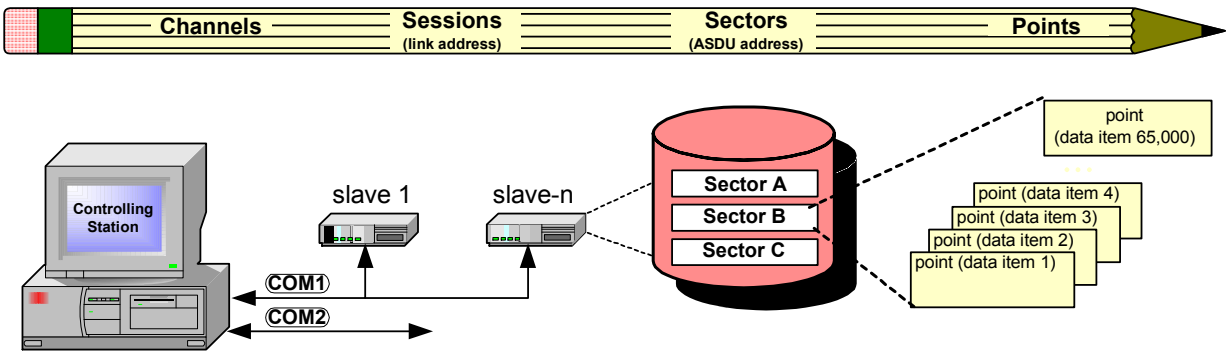

Figure 2: TMW .NET Protocol Components Data Model

3.2.3 Channels

As mentioned above, the TMW .NET Protocol Components encapsulate the physical, link, and transport layers into a single entity called a channel. Most implementations deal with channels rather than dealing directly with a specific physical, link, or transport layer implementation. Figure 3 below illustrates communications channels in a typical implementation.

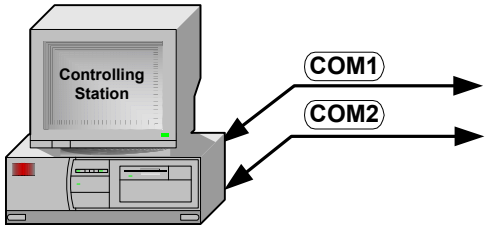

Figure 3: Illustration of Communication Channels

The TMW .NET Protocol Components exchange data with remote devices through one or more Communication Channels. The communication channels illustrated in Figure 3 are COM1 and COM2.

The communication channels may be physically discrete channels or logically discrete channels that share a physical connection. An example of a logical communication channel is a TCP/IP "serial pipe", which may share a physical 10BaseT interface with other TCP/IP serial pipes.

3.2.4 Sessions

Each channel can have one or more sessions. A session is a communication connection between the master device and one remote slave device. The total number of sessions is the total number of remote slave devices over all communication channels with which the master communicates. Figure 4 below illustrates multiple sessions in a multi-channel setup.

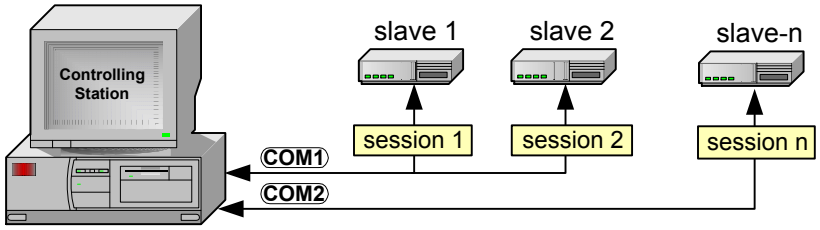

Figure 4: Illustration of Communication Sessions

When multiple sessions use the same communication channel (a multi-drop topology), the link address for the session must be unique among all sessions using that channel. In messages transmitted or received on a communication channel, the link address identifies the remote Slave device as either the destination of requests or the source of data.

Communication protocol specifications supported by the TMW .NET Protocol Components use the ISO protocol standards for the Application (7), Data Link (2), and Physical (1) Layers. It should be noted that "session" in this product and document does *NOT* refer to the defined ISO Session(5) Layer.

3.2.5 Sectors

In IEC 60870-5 protocols, remote Slave devices may contain collections of data, called sectors. DNP3 and Modbus do not support sectors. Figure 5 below illustrates a slave device that collects data into three sectors.

An example of a remote Slave device with multiple sectors is a data concentrator device that uses sectors to image downstream devices; i.e., each sector represents a single downstream device. The data within a sector represents data collected from the corresponding downstream device. Also, command requests sent by the library to a sector within the data concentrator may also result in commands being transmitted to the corresponding downstream device.

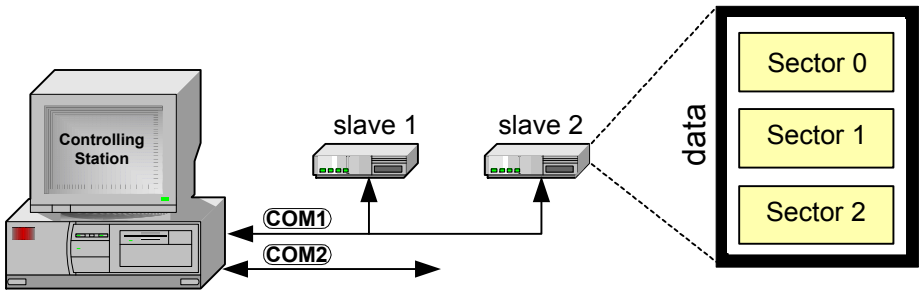

Figure 5: Illustration of Multiple Sectors

The Common Address of ASDU field may be either a single-octet or a two-octet field. It is configured for each session when the session is opened. Although it is possible for the size of the field to be different for each device (even on the same communication channel), the IEC 60870-5 protocol specifications dictate that the field size must be the same for all devices in the system (across all communication channels).

A Common Address of ASDU value of zero is not used. The value 255 (or 0xFF) for a single-octet address or 65535 (or 0xFFFF) for a two-octet address is used as a broadcast or global address in messages sent by a controlling station. A slave receiving such a message should process the message as if it were addressed to each sector (each different Common Address of ASDU) in the slave.

3.2.6 Data Points

A data point is an indivisible data object in the library and the level at which data is transferred to the target application database manager. In the IEC 60870-5 protocols, an Information Object Address is used to identify data points. The DNP3 protocol uses point numbers to identify data points.

3.2.6.1 Information Object Address

The Information Object Address field in IEC 60870-5-101 may be a one-, two- or three-octet field. The length of the IOA is configured when the session is opened. Although it is possible for size of the field to be different for each device (even on the same communication channel), IEC 60870-5 protocol specifications dictate that the field size must be the same for all devices in the system (across all communication channels).

Information Object Address zero is not assigned to any data object. All other values are permitted. If a three-octet Information Object Address is used, the address values are a sparse mapping of only 65535 different values.

Information Object Addresses are not required to be sequential. They are typically organized in a "structured" format, where each data type occupies a range of values. For example, all binary inputs could be placed in the range 2000 to 2999, and all binary outputs in the range 3000 to 3999.

In the IEC 60870-5-103 protocol, the Information Number identifies a collection of related data, so an Element Index is required to uniquely identify each data point. In

addition, a remote device can define the same information number for different protective relay "Function Types." Therefore, in the IEC 60870-5-103 protocol, three fields are required to identify each data point: Information Number, Element Index, and Function Type.

In the IEC 60870-5-102 protocol, the Record Address defines a collection of related data, so an Information Object Address is required to uniquely identify each data point. Therefore, in the IEC 60870-5-102 protocol, two fields are required to identify each data point: Record Address and Information Number.

The IEC 60870-5 protocols categorize every data point in a device into a data type. For example, in IEC 60870-5-101, "single-point information" is a 1-bit binary value, and "Measured value, normalized value" is a 16-bit integer value. Monitored inputs and controlled outputs are considered to be separate types. Hence, if a data entity could be both set (controlled) and monitored, it must be assigned two data points: a control point and a monitor point, which must have different Information Object Address values.

Example Address Map

Consider a typical small slave device, having a fixed data object mapping, as shown in Table 2.

Table 2: Example IEC 60870-5 Address Map

Data Object	Object type	Information Object Address
CB Status	Double-Point Information	1
Bus Isolator	Double-Point Information	2
Feeder isolator	Double-Point Information	3
Bus Earth Switch	Single-Point Information	4
Feeder Earth Switch	Single-Point Information	5
Battery Fault	Single-Point Information	6
Control Supply	Single-Point Information	7
Auto Reclose In Progress	Single-Point Information	8
Bus A-Phase KV	Measured Value, Scaled	75
Bus B-Phase KV	Measured Value, Scaled	76
Bus C-Phase KV	Measured Value, Scaled	77
Feeder A-Phase KV	Measured Value, Scaled	78
Feeder B-Phase KV	Measured Value, Scaled	79
Feeder C-Phase KV	Measured Value, Scaled	80
A-Phase Amps	Measured Value, Scaled	81
B-Phase Amps	Measured Value, Scaled	82
C-Phase Amps	Measured Value, Scaled	83
CB Control	Double Command	128
Bus Isolator Control	Double Command	129
Feeder Isolator Control	Double Command	130
Bus Earth Switch Control	Single Command	171
Feeder Earth Switch Control	Single Command	172
Auto Reclose Enable	Single Command	173
Trip Current	Set Point Command, Scaled	190

Note from this example mapping:

- Each Information Object Address value is unique. The values may be assigned in any order, and do not need to form a contiguous numeric sequence.
- In an IEC 60870-5 protocol, message efficiency is maximized if objects with the same object type are assigned consecutive Information Object Address values.
- If an association is to be established between a controllable object and a monitored object, they must have corresponding object types (e.g., Single Command corresponds to Single Information; Set Point Command, Scaled Value corresponds to Measured Value, Scaled Value; etc.).
- For instance, in the example mapping, the CB Status (Double-Point Information, Information Object Address 1) and CB Control (Double Command, Information Object Address 128) could be associated. When the

Master issues a command to Information Object Address 128, it would expect to see a corresponding change in the value of Information Object Address 1. If the CB Control had been defined as a single command, this association would not be possible because the control object type would not match the monitored object type.

- For a master to operate with this slave device, both master and slave must be configured to use the same Data Link Address and Common Address of ASDU, and the data object types and Information object Address values of the slave device must be configured in the Master. The master is also configured to associate any controlled objects and their corresponding indications, as appropriate.
- If two identical devices of this type were to be used in a system, they must be assigned different Common Address of ASDU values, to conform to the requirement that the combination of Common Address of ASDU and Information Object Address is unique across the whole system.

3.2.6.2 DNP3 Point Numbers

DNP3 uses a point number to identify a specific data point. Point numbers start with 0 for each data type, and can go up to 65535. Unlike the IEC 60870-5 protocols, point numbers are associated with the data type, so a given number can be used in multiple data types.

The DNP Technical Committee strongly recommends using contiguous point numbers, starting at 0 for each data type because some DNP3 Master implementations allocate contiguous memory, from point 0 to the last point number for each data type.

While we do not recommend this approach in new DNP3 Master designs, we suggest that a DNP3 Slave device be designed with a contiguous memory map to avoid wasting memory in some DNP3 Master implementations.

In some implementations, it may be desirable to have small gaps in point numbers. Such gaps are easily accommodated by the TMW .NET Components by setting the “missing” SDNPSimPoint to Enabled = false.

3.2.6.3 Modbus Point Numbers

Modbus uses a point number to identify a specific data point. Point numbers start with 0 for each data type, and can go up to 65535. Unlike the IEC 60870-5 protocols, point numbers are associated with the data type, so a given number can be used in multiple data types.

It is strongly recommended that contiguous point numbers be used, this will improve communications efficiency.

We suggest that a Modbus Slave device be designed with a contiguous memory map to maximize performance.

In some implementations, it may be desirable to have small gaps in point numbers. Such gaps are easily accommodated by the TMW .NET Protocol Components.

3.2.7 Target Interface

The target interface provides the interface between the TMW .NET Protocol Components and the target hardware and software. This interface is implemented in the WinIoTarg.dll and should not need modification. In the future we may provide mechanisms to allow the modification of the Target Interface for TMW.NET.SCL based applications. Note that with the source code version of the TMW .NET Protocol Components modification of the target interface is possible.

3.2.8 Database Interface

The database interface provides the interface between the TMW .NET Protocol Components and the target systems data. For a slave device this will typically be tied directly to the actual data points in hardware. For a master device this will typically interface with a database of some sort.

The database interface consists of several events that may be registered to interface to the actual target database. In general there will be several events for each supported data type. Events for data types that will not be supported on the target platform do not need to be implemented.

3.2.9 Application Interface

The TMW .NET Protocol Components Application Interface consists of methods called by the application to instruct the TMW .NET Protocol Components to perform specific operations. For a slave device this is limited to opening and configuring channels, sessions, and sectors, and possibly generating events. For a master device this includes opening and configuring channels, sessions, and sectors, and issuing requests to be sent to the remote devices.

In addition to calling into the TMW .NET Protocol Components, the application interface includes a number of events that can be used to invoke functions within the target application when events take place in the TMW .NET Protocol Components. These events are called when a request completes, an application layer fragment is received, a statistically significant event occurs, etc. Each of these will be discussed in more detail below.

3.2.9.1 Events

The TMW .NET Protocol Components supports a number of events defined in the Application, Channel, Session, Sector, Request and Database classes to allow the TMW .NET Protocol Components to tell the users application that certain events have occurred. Some examples are:

1. Opening, closing, modifying Channels, Sessions, and Sectors.

2. Communications and Statistics
3. Database changes
4. Request completion

Although the sections below describe the class of events supported by the TMW .NET Protocol Components it is recommended that you refer to the Help files for details.

3.2.9.1.1 Channel, Session and Sector Events

Channels, Sessions and Sectors all have open, close and modify events. The user application can register for these events to get notification when a Channel, Session or Sector is opened, modified or closed. Typically these events are used to synchronize the user application with the TMW .NET Protocol Components. These events occur when the Open, Close or Modify methods are called on these objects.

3.2.9.1.2 Communications and Statistics Events

The Channel, Session and Sector have Statistics events available to notify the user application when certain status and error values maintained by the TMW .NET Protocol Components change. In addition the Channel has some communications events available to notify the application when data receipt or transmission occurs.

3.2.9.1.3 Database Events

A database has two modes of operation.

1. Built in database
2. User defined database

When using the built in database, the TMW .NET Protocol Components will notify the user application of database changes by calling the Add, Update, Delete events on the database object. If UseSimDatabase is set to false, the database will also notify the user application of database changes by calling various store, get, validate, etc. events.

Here are the supported events from TMWSimDatabase that occur even when using the built in database:

```
// Summary:
//     Event is generated when a database point is added
public event TMWSimDataBase.AddPointEventDelegate AddPointEvent;
//
// Summary:
//     Event is generated when the database is cleared
public event TMWSimDataBase.ClearDBEventDelegate ClearDBEvent;
public event TMWSimDataBase.CloseDelegate CloseEvent;
//
// Summary:
//     Event is generated when a database point is deleted
public event TMWSimDataBase.DeletePointEventDelegate DeletePointEvent;
//
// Summary:
//     Event is generated when a database point changes
public event TMWSimDataBase.UpdateDBEventDelegate UpdateDBEvent;
```

3.2.9.1.4 User Defined Database Interface

The master database interface is designed to support the storage of values returned from slave devices. In some cases, the data points will be known by the master, in other cases the master database should support the dynamic creation of points returned from the slave device.

A separate database is maintained per session for DNP and Modbus or per sector for the IEC 60870-5 protocols. When the Session or Sector is opened the TMW .NET Protocol Components call a database initialization event to allow the target application to initialize its database. The target application may then set the Tag property of the TMW .NET Protocol Components database to something meaningful to the application, such as a pointer, index or other reference. The TMW .NET Protocol Components will provide this Tag property through the TMW .NET Protocol Components database with all future events on the database.

The TMW .NET Protocol Components provide an individual data store event for each type of data point object. The input parameters to these events include the Database, the Information Object Address that identifies the data point, and the data values that need to be stored.

A database close event is called by the TMW .NET Protocol Components so that the target application can perform whatever actions are required by its database when the session or sector is closed.

3.2.9.1.5 Request Completion Events

As a master, an application can issue commands or requests to a slave. When these requests are issued event handlers can be specified that notify the user application when the requests complete.

Here is an example request completion event from MDNPRequest:

```
/// Register to receive an event on command response or when completion
delegate void RequestEventDelegate(MDNPRequest ^request,
    MDNPResponseParser ^response);
event RequestEventDelegate^ RequestEvent;
```

3.2.10 Diagnostics

The TMW .NET Protocol Components support the generation of diagnostic information that can be used to analyze the protocol message traffic. All TMW .NET Protocol Components diagnostic information is passed to the user application through the ProtocolBuffer class. This class manages a list of ProtocolDataObjects that contain the diagnostic message and a structure that identifies the source of the message. This message source can be used to filter or redirect the diagnostic information.

The following snippet of code illustrates use of the diagnostics:

```
private void ProtocolEvent(ProtocolBuffer buf)
{
  for (int i = buf.LastProvidedIndex; i < buf.LastAddedIndex; i++)
  {
    Console.WriteLine(buf.getPdoAtIndex(i).ProtocolText);
  }
}

protocolBuffer = TMWApplicationBuilder.getProtocolBuffer();
protocolBuffer.ProtocolDataReadyEvent += new
ProtocolBuffer.ProtocolDataReadyEventDelegate(ProtocolEvent);
protocolBuffer.EnableCheckForDataTimer = true;
```

The above code registers an event that provides the diagnostic data to the user application. The ProtocolEvent handler will be called periodically as protocol diagnostic data becomes available. Note that this utilizes a windows forms timer (i.e. System.Windows.Forms.Timer) as a result this mechanism only works in a windows forms application.

To use the Protocol Buffer in a non windows forms application (i.e. console or windows service) do the following.

```
    static private bool bProtocolMode;
    static ProtocolBuffer protocolBuffer;
    static System.Threading.Timer protocolTimer;

    static void OnUpdateProtocolBufferTimer(object obj)
    {
      protocolTimer.Change(System.Threading.Timeout.Infinite,
         System.Threading.Timeout.Infinite);
      protocolBuffer.DoForceUpdate();
      protocolTimer.Change(500, 500);
    }

    static void OnNewProtocolData(ProtocolBuffer buf)
    {
      buf.Lock();
      for (int i = buf.LastProvidedIndex; i < buf.LastAddedIndex; i++)
      {
        string text = protocolBuffer.getPdoAtIndex(i).ProtocolText;
        Console.Write(string.Format(">>>Protocol: {0}", text));
      }
      buf.UnLock();
    }

    if (bProtocolMode)
    {
       protocolTimer = new System.Threading.Timer(OnUpdateProtocolBufferTimer, null,
         500, 500);
       protocolBuffer = TMWApplicationBuilder.getProtocolBuffer();
       protocolBuffer.ProtocolDataReadyEvent += OnNewProtocolData;
    }
```

In the above code instead of letting the ProtocolBuffer fire the event indicating new protocol data is available. It is up to the user application to perform this through the

protocolBuffer.DoForceUpdate() call. This mechanism is implemented in the example by default located at

C:\Program Files\Triangle MicroWorks\.NET Protocol Components\Samples\DNP\sampleApps\DNPDatabaseEvents

3.2.10.1.1 Diagnostic Memory Usage

The diagnostics functionality described above is enabled by default. The ProtocolBuffer object will maintain a potentially large number of diagnostic messages, allowing you to "scroll back" through a large number of stored messages. By default this value may be quite large (for example 10,000). Depending on your needs, you can turn off the ProtocolBuffer logging functionality, reduce the number of messages that are stored, or periodically clear the buffer to free up that memory.

To disable the protocol logging functionality completely, when you create the TMWApplicationBuilder object you can pass it the argument "false". If you do this you must NOT try to use the ProtocolBuffer object, it will be NULL.

```
applBuilder = new TMWApplicationBuilder(false);
//ProtocolBuffer protocolBuffer = TMWApplicationBuilder.getProtocolBuffer();
```

To reduce the number of messages that are buffered you can set the MaxSize property on the ProtocolBuffer object.

```
protocolBuffer.MaxSize = 100;
```

To cause the buffered message objects to be deallocated you can call the Clear methodon the ProtocolBuffer object.

```
protocolBuffer.Clear();
```

3.2.11 Condition Notification and Statistics

The TMW .NET Protocol Components generate events that can be monitored by the user. These events are generated whenever something significant happens in the component. This includes any error, transmission or reception of bytes, frames, and fragments, as well as others. These events can be used to maintain statistics, invoke additional processing, or whatever is required by the user's implementation.

This feature is exposed through the statistics events available on the channel and session.

3.2.12 Asynchronous Database Updates

Database updates on master devices can frequently take a considerable amount of time. This is typically the case when updating a relational, or otherwise complex, database. If the amount of time required to update the master database precludes returning to process input data and/or respond to requests in the TMW .NET Protocol Components in a timely fashion the database updates must be handled asynchronously to the TMW .NET Protocol Components processing.

The TMW .NET Protocol Components ships configured for asynchronous database accesses. In this mode, the TMW .NET Protocol Components stores data on a database queue.

Synchronous database accesses are not currently not supported in the binary versions of the TMW .NET Protocol Components.

3.3 Master Overview

This section describes details specific to Master implementations.

3.3.1 Architecture

This section describes architectural details specific to a master implementation. In addition to the basic features and functionality available in all TMW .NET Protocol Components, each Master Library performs the following functions:

1) Generate and send requests to remote devices as directed by the user application through the associated request objects.
2) Monitor for messages from slave devices and process as follows:
    a. The message is first checked to see if it is in response to an outstanding request. If so, it processes the request, calling any user events and either closes the request or proceeds to the next step of the request.
    b. If the message was an 'unsolicited' message it is processed and the appropriate action is taken.

3.3.1.1 Command Interface

The master command interface allows target application to generate and send requests to remote devices. The interface is provided in the TMWRequest class hierarchy. There are specific individual request methods provided for each command type. Most of the methods require additional arguments depending on their specific purpose.

For each request, the target application can specify an event to be called upon completion of that command. The target application can also specify a timeout value for each command request.

Each request function returns Boolean to indicate whether the request was sent or queued for transmission (true) or was not queued (false). The request object should be used for future reference to this outstanding request. For example, the object has a method that could be used to cancel this request.

An interface is also provided to register to receive unsolicited or asynchronous messages from a remote device. This allows the target application to receive messages on a per session basis in addition to the responses to commands that were sent.

3.3.2 Supporting Redundant Communication Paths

There are several mechanisms for supporting redundant communications paths to an outstation device. The following paragraph summarizes the most common redundancy schemes.

1) The master simultaneously collects data over each channel. The same data are reported over each channel, and the Master merges these duplicated event reports into a single database update stream.

   This method can be implemented by creating a separate session for each redundant path. When data is reported, the TMW .NET Protocol Components call the database interface routine for the current session. The target's database interface routine (or some higher level process) is responsible for processing duplicate events and merging the data into a single stream of database updates.

2) The master collects data from a single one of the possible paths to the slave, and switches to another path if the original channel fails. In this case, Master monitors the channels switches channels as appropriate.

   One way to implement this method is to switch channels when a command fails. For example, the request method that initiates a request message can be configured to call an event when the command completes.

   Upon completion, the TMW .NET Protocol Components call the specified event passing a status indication. The event handler can, upon receipt of a failure indication (e.g., FAILURE or TIMEOUT), reconfigure the device for a different channel, and resubmit the request message. In most systems, it is also desirable to notify the end user that this switch has taken place.

   With this method, it is usually desirable to maintain a concept of primary and secondary channels, as the secondary channel is likely to be a slower and/or more expensive connection. Thus, it may be desirable to implement logic to switch back to the primary channel when it becomes available again.

Note that with either method, it is important to periodically verify the status of the redundant channel. With method 1), user notification can be made whenever either channel fails. With method 2), it is usually desirable to perform periodic link checks on the backup channel. The DNP3 link status message (configured via the LinkStatusPeriod parameter of the MDNP Session object) can be used for this purpose.

3.4 Slave Overview

This section describes details specific to Slave implementations.

3.4.1 Architecture

This section describes architectural details specific to a slave implementation. In addition to the basic features and functionality available in all TMW .NET Protocol Components each slave TMW .NET Protocol Component performs the following functions:

1) Wait for and process requests from the remote master devices.
2) Optionally scan for change events.
3) Transmit events, either unsolicited or when requested.

3.4.1.1 Event Generation

The Slave Protocol Component can be configured to periodically scan for changes in data points and automatically generate events, or user provided code can explicitly call the appropriate AddEvent method, when it determines that something of significance has occurred. If the periodic scan option is chosen, the user can specify how often to scan for changes on a per session or per sector basis depending on the protocol by setting xxxScanPeriod to a nonzero value. See the sample code for examples of scanning or calls to the AddEvent methods.

3.4.2 Supporting Redundant Communication Paths

There are several mechanisms for introducing redundancy in an outstation (slave) device. The following paragraph summarizes the most common redundancy schemes.

DNP3, IEC 60870-5, and other modern SCADA protocols employ an event reporting paradigm that ensures that field events that occur between data polls are reported to the master. To provide this feature, event buffers are created in the slave, and these buffers are reported to the master. If multiple redundant communication paths are provided between the master and the slave, the method of reporting events must be coordinated between these paths. This can be done in any of the following methods:

1) The master simultaneously collects data over each channel. The same data are reported over each channel, and the Master merges these duplicated event reports into a single database update stream.

   This method can be implemented by creating a separate session for each redundant path. When an event occurs, the target code should call AddEvent for each data type. Each session or sector will then manage its own event queue to ensure that the event is passed to the Master.

2) The master collects data from a single one of the possible paths to the slave, and switches to another path if the original channel fails. In this case, Master monitors the channels and switches channels as appropriate.

One way to implement this method is to write tmwtarg_receive() to accept data from both a primary and secondary communication port and record the last port on which data are received. The tmwtarg_transmit() function sends data on the last port from which data were received.

With this scheme, the Library is only aware of one session on one channel; however, the target hardware is actually managing the two redundant serial ports transparently.

This method can only be implemented in the Source code version of the TMW .NET Protocol Components by the end user and requires that WinIoTarg.dll be modified.

Chapter 4 TMW .NET Protocol Components Source Code Version

This section describes the specifics related to the Source Code version of TMW .NET Protocol Components. These sections discuss the steps required to install and build the TMW .NET Protocol Components.

Of course everything in the preceding sections about the functionality, architecture and interfaces to the TMW .NET Protocol Components applies to the Source Code Version. There are some differences in the directory structure and location of the samples files because of the presence of the source code.

4.1 Unpacking and Installing

TMW .NET Protocol Components Source Code version is delivered as self extracting executable file. This section describes in detail how to install and build the component.

4.1.1 Installation

To install the source code, simply run the self extracting executable to install in the desired directory.

4.2 Building

Once the component is installed, the first step should be to build the component as delivered using Visual Studio 2008. The TMW .NET Protocol Components are written using ANSI C, C++/CLI and C# so this is typically a straightforward process.

All of the TMW .NET Protocol Component files will be installed under the directory specified during installation. The files will be installed in one or more subdirectories of the installation directory. The directories will include a number of Visual Studio project file as well as a single solution file for the entire component. This solution file can be found at <installdir>/tmwscl/tmw.scl.sln.

Simply load this Visual Studio solution file into Visual Studio 2008 and perform a 'Rebuild All'. This will build the samples as well as the TMW .NET Protocol Components.

The source code versions contain ANSI C source code that implements the main protocol stack. Although rarely required please refer to the Source Code Library Manual for details on how to modify or debug the ANSI C source code. The source code versions also contain managed C++/CLI and C# source code for use in .NET environments. Your application should be written to make use of the interface provided by this managed code. The files that implement for the managed source code classes resides in <installdir>/tmwscl/.NET/tmw.scl

4.3 Channels, Sessions, and Sectors

Each protocol has a channel class derived from the base class TMWChannel. For example DNPChannel provides the interface to the DNP Channel functionality.

MBChannel, FT12Channell, I102Channel etc. provide the interface to the other protocols. Each protocol also has a master and slave session class derived from TMWSession. For example, M101Session provides the interface to the Master 101 session. Additionally, the IEC protocols have a master and slave sector class derived from TMWSector.

4.4 Database

Each master and slave session or sector contains a database interface and provides a simple database implementation. The <*scl*> SimDatabase classes (ie SDNPSimDatabase) provides the interface to the existing implementation. The event interface to provide your own database implementation is contained in <*scl*>Database (ie SDNPDatabase). See the sample code for examples of how to interface with the existing database or how to use .NET events to provide your own database implementation.

4.5 Master Libraries

This section covers porting details specific to a TMW .NET Protocol Components Master.

4.5.1 Issuing Commands

The interface to the master commands is through the classes derived from TMWRequest. Each protocol has its own <scl>Request class that provides methods for the individual requests which are supported.

4.6 Slave Libraries

This section covers porting details specific to a TMW .NET Protocol Components slave.

4.6.1 Supporting Change Events

A number of the data types derived from TMWSimPoint contain an AddEvent method, which be used to queue change events which send messages to the master indicating the new data values. To have more control over the event data or if the simple built-in database is not being used methods on the session or sector can also be used to add events, for example SDNPSession::BinInAddEvent(..) or S14Sector::MspAddEvent(..); See the sample code for examples.

4.7 Testing your Implementation

For details on how to best test your application see Chapter 6.

Chapter 5 Advanced Topics

This section discusses how the TMW .NET Protocol Components architecture can be leveraged to provide a wide range of custom solutions.

5.1 Supporting Multiple Protocols

Any of the TMW .NET Protocol Components can be combined to provide support for multiple protocols in a single device. Since the different Protocol Components share many files it is imperative that they be the exact same version. Hence the first step in supporting multiple protocols in the same device is to acquire the same version of each of the required Protocols.

After confirming that all required libraries are the exact same version, install the first Protocol into the target directory. Extract the next Protocol into the same directory. Repeat this last step for each desired protocol.

Now add the protocol specific project file to the solution.

A single target implementation in *WinIoTarg.dll* will be shared by all the Protocols. A database interface must be provided for each Session or Sector independently.

The remainder of the integration should be performed exactly as if you were integrating to the two Protocols independently. Specifically you open channels, sessions, and sectors exactly as you would if you were integrating to each Protocol independent of the others.

Note: You can create a Peer implementation by combining a Master and Slave of the same protocol.

5.2 60870-5-104 Redundancy

The TMW .NET Protocol Components architecture provides redundancy as specified in version 2 of the IEC 60870-5-104 protocol, formerly known ‘Norwegian User Conventions Redundancy’. This supports multiple redundant TCP/IP connections contained in a single IEC 60870-5-104 redundancy group. All user communications are performed on the current ‘active’ channel but each channel is monitored to make sure it is still operational. If the ‘active’ channel goes down communications are automatically switched to the next operational channel. The system then attempts to reestablish the channel that failed in addition to monitoring the other channels. The software architecture is depicted in the figure below.

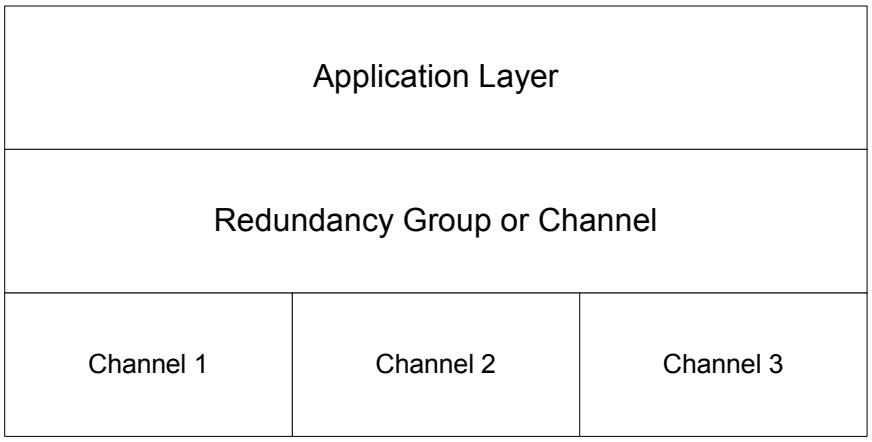

Figure 6: 60870-5-104 Redundancy Software Architecture

In this situation the physical and link layers still execute on the same processor as the application layer, but there are multiple physical/link layers or channels. A redundancy group or channel is implemented that manages multiple channels. When a request is received from the application layer it is dispatched to the current 'active' link layer. When information is received on the active link layer it is passed through the redundancy group and forwarded to the application layer. See the I104RedundantMasterGUI or I104RedundantSlaveGUI examples for more details.

5.3 DNP3 Data Sets

Data Sets provide a method for transferring structured data within the DNP3 protocol. Full support for Data Sets is provided in the MDNP and SDNP TMW .NET Protocol Components. For more information on Data Sets see the latest version of DNP Technical Bulletin TB2004-004.

Data Sets consist of Data Set Prototype and Descriptor objects, each containing a sequence of elements describing the data contained in Data Set Present Value or Snapshot Event objects. Each descriptor or prototype element consists of a descriptor code, data type code, maximum length and ancillary value. Prototypes are identified by UUIDs and can be contained by more than one Descriptor object. The actual Data Set data objects (Object Group 87 or 88) contain only a sequence of elements containing length and value. The Descriptor and Prototypes (Object Group 85 and 86) must be present on a device in order to make sense of the received Data Set data.

Data Set Static Data Objects can be read and written by a DNP3 Master and Data Set Event Objects can be sent by an Outstation spontaneously or in response to a specific or class read request.

The MDNP TMW .NET Protocol Components provide three request functions; `WriteDataset, WriteDatasetProto and WriteDatasetDescr,` to allow writing the Data Set objects to the slave. For each of these, the data elements to be sent to the outstation are retrieved using events on the database. The database events required are; StoreDatasetxxx, DatasetProtoxxx, DatasetDescrxxx and Datasetxxx events to retrieve

and store Data Set information from/to the database. The existing request methods; ReadGroup and ReadPoints and AssignClass also apply to Data Sets.

The SDNP TMW .NET Protocol Components SDNPSession class provides the AddEvent method to add Data Set Snapshot events. The sdnp database interface requires DatasetProtoxxx, DatasetDescrxxx and Datasetxxx events to read and write Data Set information from/to the database.

Data Set Descriptor Object Group 86 supports 3 variations. The information transferred in these 3 variations is different. Variation 1 which can be read or written, describes the data contained in a Data Set. Variation 2, characteristics, which can be read but not written, indicates whether a Data Set can be read or written, if outstation maintains static and or event Data Sets and if the Data Set was defined by the master or outstation.

5.4 IEC 60870-5-101/4 Control Points

In IEC 60870-5-101 and -104, Monitored Points (ie Single Type T1) are read-only. Similarly, Point Commands (ie Single Type T45) are write-only. In the simple built-in default database provided with the .NET Protocol Components, command points are addressed by adding 2000 to the corresponding monitor point Information Object Address (IOA). To write to the point monitored by Single Point with IOA 100, a write should be done to the Single Point Command Point with IOA 2100.

The offset for control points in the built-in database can be changed using the method S14SimDatabase::SetCtrlOffset(..);

5.5 60870-5-101/4 Time Stamps.

TMWApplication::TimeMode can be set to SIMULATED, SYSTEM or SYSTEM_NO_SETTIME. This controls the general time behavior of the .NET Components library. When your application code wants to get the current time it should generally call S14Sector::GetTimeStamp() to get the time for a particular sector. Depending on what TimeMode you are using and if clock syncs are being received on multiple sectors, the time could be different.

If you set S14Database::UseSimDatabase to false you can also register for two time related .NET database events: CcsnaGetTimeEvent and CcsnaSetTimeEvent. The library will get and set times according to TimeMode above, but will then call the registered .NET events allowing your target to also set and get these times. If you don't implement these two events, the time will be used according to TimeMode above. If you do register these two events you can handle time as you wish.

There are other .NET database events such as MboGetValueFlagsTimeEvent that also allow you to specify a timestamp for individual responses. You should return the time that suits your implementation, but generally you should call S14Sector::GetTimeStamp() to get the time for that sector, or return the same time you would return from CcsnaGetTimeEvent if you registered it (since GetTimeStamp() will call that method to get the time for the sector).

There are other .NET component methods that require a time stamp as an argument. You can pass any time that suits your application, but generally you should call S14Sector::GetTimeStamp() to get the time for that sector.

5.6 60870-5 Private or Custom ASDU Support

The .NET Components implement standard ASDUs documented in the appropriate standard documents. The 60870-5 protocols allow for private or custom ASDUs using other private range type ids. If you have purchased the Source Code version of the .NET components you can implement custom ASDUs by copying and modifying code from standard ASDUs. If you are using an executable version of the .NET Components you can also send and process Custom ASDUs. The i10xMasterGUI and i10xSlaveGUI samples provide examples of how to implement custom ASDUs for these protocols.

5.6.1 Master

To send a Custom ASDU request you would call the MxxRequest::CustomAsdu() method, providing the values for the Data Unit ID portion of the message as well as the rest of the octets that make up the request. You also have the option of specifying the entire byte sequence of the message including the Data Unit Id by setting the first parameter dataUnitIdInData to true.

To process a response that contains a private Type Id, you would register a M1xxSession.ProcessCustomASDUEvent. This method would be called whenever an application layer response is received on the master. This method can decide whether to process this response and return true, or return false allowing the .NET Component library to process the response. Even standard response ASDUs can be intercepted by this method by returning true.

5.6.2 Slave

To process Custom ASDUs you must register two events; S1xxSession::ProcessCustomASDUReqEvent and S1xxSession::BuildCustomASDURespEvent. The Process event will only be called if an Unknown Type Id is received by the slave. If this method returns true the library will assume the request has been handled, if this method returns false the library will treat this as an unknown type id and process it according to the protocol.

The Build event will be called if a class 1 response was requested by the master and there are no other class 1 responses to send. This function should return true if it needs to send a custom ASDU response and if the buildResponse parameter was true it should also build and send the response by calling S1xxSector::SendCustomASDUResponse.

5.7 SSL or TLS over TCP/IP

The .NET Components allow you to use Secure Socket Layer (SSL) or Transport Layer Security (TLS) over TCP connections. TLS is an IETF standard based on Version 3 of SSL. Using these allows for verification of the server and or client credentials as well as encryption of the data transferred. Because of Export/Import restrictions, the SSL/TLS negotiation and encryption/decryption code is not implemented or contained in the .NET Protocol Components. Instead the TMW .NET Components make use of a readily available Open Source toolkit, OpenSSL, to provide that functionality. See http://openssl.org/ for more information about that project.

To open a TCP channel using SSL/TLS the OpenSSL libraries must be first installed on your PC. You can either download the OpenSSL source code and build your own Windows version of the libraries, or you can find precompiled libraries available for download to your PC (search "openssl for windows" in your internet search engine).

The .NET Component executables require ssleay32.dll and libeay32.dll, if a channel is opened using SSL/TLS. These DLLs, should usually be placed in the same directory as the executable to make sure they are found by the executable. The version and compile flags of these DLLs must match the Runtime Libraries that the .NET Component Library was built against. For .NET Components version 3.9.5, the OpenSSL version 1.0.0a, DLLs, built with /MD (Multithhreaded DLL) is required. If you have the source code version of the .NET Components you can of course rebuild them using the debug version of the libraries if desired.

If you want to build your own version of the OpenSSL source code you should download the appropriate version and follow the build directions for Windows. Here is an example procedure that we used. You can read the OpenSSL INSTALL.W32 or search the internet for more details or options.

```
Install perl and nasm (if you do not already have them on your PC)
cd yourpath\openssl  (both here and below replace "yourpath" with the correct path)
SET PATH=%PATH%;C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin
SET LIB=%LIB%;C:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib
SET INCLUDE=%INCLUDE%;C:\Program Files\Microsoft SDKs\Windows\v6.0A\Include
perl Configure VC-WIN32 no-asm --prefix=yourpath/openssl
ms\do_ms
nmake -f ms\ntdll.mak
```

The above should create a directory out32dll that contains the 2 required DLLs. These DLLs should be copied to the correct executable directory.

OpenSSL also requires parameters to initialize the Diffie-Helman key exchange algorithm. These are placed in a file named dh1024.pem. Examples of this file are available along with the OpenSSL distribution. The OpenSSL documentation says "the risk in reusing DH parameters is that an attacker may specialize on a very often used DH group. Applications should generate their own DH parameters during the installation process using the openssl dhparam(1) application".

The sample applications DNPMasterGUI and DNPSlaveGUI contain code required to configure and enable SSL/TLS over TCP. Note that this code is commented out in the samples. The names of files containing certificates and other data required by OpenSSL should be modified to use your actual values.

```
// SSL/TLS Enabled on this channel.
// true requires ssleay32.dll and libeay32.dll from OpenSSL
// Also requires Diffie-Helman dh1024.pem file and other files
// which may be specified here.
slaveChan.SslTlsEnabled = true;

// file containing root certificate authority certificate
slaveChan.SslTlsCertAuthFile = "root.pem";

// file containing server certificate
slaveChan.SslTlsCredentialsFile = "key.pem";

// password for private key in credentials file
slaveChan.SslTlsPassword = "yourPassword";

// If false, even if SSL Credentials are not valid connection will be allowed.
// If true, if SSL Credentials are not valid if SslTlsCheckCertificateEvent
// is registered it will be fired. If this event is not registered or the event
// returns false the connection will be broken.
slaveChan.SslTlsVerifyCert = true;

// true, if you want to verify the master's credentials
slaveChan.SslTlsVerifyClient = false;

// event to be fired if Credentials are not valid.
slaveChan.SslTlsCheckCertificateEvent += new
TMWChannel.SslTlsCheckCertificateEventDelegate(slaveChan_SslTlsCheckCer
tificateEvent);
```

When you run your application, you should look at the diagnostics messages in the protocol log from the target layer to determine if things are working properly. You should see messages similar to the following for a DNP Slave (Outstation) indicating that SSL is configured.

```
### .NET DNP Slave - 127.0.0.1:20000 - TCP open
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Initialized library
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Created context
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Initialized Context
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Opened DH file
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Set DH parameters
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Read certificate file key.pem
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Read key file key.pem
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Read CA list root.pem
### .NET DNP Slave - 127.0.0.1:20000 - TCP Listen, successfully listening
```

Sometime later when a connection from the master arrives you should see messages similar to the following.

```
### TCP LISTENER: accept incoming connection
### .NET DNP Slave - 127.0.0.1:20000 - TCP LISTENER: SSL connection accepted
### TCP LISTENER: listenThread connection accepted from 127.0.0.1
```

If you see the following message, verify that the two required OpenSSl DLLs are present in the executable directory, or in a directory that can be found by the executable.

```
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, load ssleay32.dll failed
```

If you see the following messages, verify the names and location of the files configured. They should normally be placed in the same directory as the executable.

```
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Can't read certificate file key.pem
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Can't read key file key.pem
### .NET DNP Slave - 127.0.0.1:20000 - TCP SSL, Can't read CA list root.pem
```

If you get an error similar to the following when you try to run your application, it means that the DLLs you are using are not compiled with the same configuration as the .NET Components. In particular you might be trying to use a debug version instead of a release version of the DLLs.
OPENSSL_Uplink(0AA69000,08): no OPENSSL_Applink

When the TCP connection is made and SslTlsEnabled is true, the OpenSSL library will try to verify the credentials. By default if this verification fails, the SslTlsCheckCertificateEvent will be fired if it is registered, or if not the connection will be broken. The event if registered can decide whether to allow the connection anyway based on the OpenSSL error such as X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN, passed to it or by further evaluating the OpenSSL structure that is passed to it. The second parameter contains a pointer to a SSL structure. You may choose to call OpenSSL functions to get information about what is contained in that structure. If SslTlsVerifyCert is set to false, the connection will be allowed even if the credential check fails.

If the SSL/TLS connection is made between two hosts it is possible to view the data being exchanged using WireShark or a similar TCP analyzer program. Using port 443 allows WireShark to provide a more detailed SSL/TLS interpretation.

5.8 Installation on Target PCs

When the .NET Protocol Components are installed on your machine, the install from TMW checks for required software from Microsoft as well as installing the executables themselves from TMW. If you have purchased a Redistributable License or a Redistributable License with Source Code you will need to create your own install package or procedure.

Our software requires the following downloads from Microsoft

Microsoft .NET Framework Version 2.0 Redistributable Package (x86)

Microsoft .NET Framework 2.0 Service Pack 1

Microsoft Visual C++ 2008 SP1 Redistributable Package (x86)

Microsoft Visual C++ 2008 Service Pack 1 Redistributable Package ATL Security Update

In Visual Studio InstallShield they show up as

Visual C++ 9.0 MFC (x86) WinSXS MSM 9.0 Merge Module (9.0.30729.1)

Visual C++ 9.0 CRT (x86) WinSXS MSM 9.0 Merge Module (9.0.30729.1)

Visual C++ 9.0 ATL (x86) WinSXS MSM 9.0 Merge Module (9.0.30729.1)

Visual C++ 9.0 MFC Policy (x86) WinSXS MSM 9.0 Merge Module (9.0.30729.1)

Visual C++ 9.0 CRT Policy (x86) WinSXS MSM 9.0 Merge Module (9.0.30729.1)

Visual C++ 9.0 ATL Poilicy (x86) WinSXS MSM 9.0 Merge Module (9.0.30729.1)

Visual C++ 9.0 MFC (x86) WinSXS MSM 9.0 Merge Module (9.0.30729.4148)

Visual C++ 9.0 CRT (x86) WinSXS MSM 9.0 Merge Module (9.0.30729.4148)

Visual C++ 9.0 ATL (x86) WinSXS MSM 9.0 Merge Module (9.0.30729.4148)

Microsoft .NET Framwork 2.0 SP1 InstallShield Prerequisite

You would need to also provide the following libraries from TMW or you may build them if you have purchased Source Code from TMW.

TMW.SCL.dll

WinIoTarg.dll

TMW.BugReporter.dll

TMW.SCL.XMLParser.dll

SCLLicenseManager.dll (only if redistributable or single use. Not required for Source Code Library executables)

The following two libraries are also required if OpenSSL is enabled in TH and .NET 3.9.5 and later.

ssleay32.dll

Libeay32.dll

DNPMaster.exe (for example, whatever the executable is named.)

Chapter 6 Debugging and Testing your Implementation

This chapter will discuss techniques and tools which can be used to test and debug the TMW .NET Protocol Components on the target device. The information in this chapter consists of techniques that have proven useful in previous situations. Each device and implementation is different and it is up to the user whether these techniques are appropriate or not.

It is recommended that testing begins as soon as the device is able to communicate and continue incrementally until all the desired features are complete. This allows problems to be found before they are built upon and potentially made more complex. It also allows the user to see progress from the very early stages of the development.

The examples in this chapter make use of the Triangle MicroWorks Inc. Protocol Test Harness. The Protocol Test Harness is a Windows application that supports the testing of master or slave devices for all the protocols supported by the TMW .NET Protocol Components. Use of the TMW Protocol Test Harness is not required, but a system with similar features is highly desirable. A 21 day evaluation version of the TMW Protocol Test Harness can be downloaded from the Triangle MicroWorks, Inc. web site at http://www.TriangleMicroWorks.com/downloads.htm

6.1 Testing Basic Communications

Basic communications can be tested as soon as the TMW .NET Protocol Components user application has the ability to establish a channel, session, and/or sector. The recommended approach to testing basic communications is to open a single channel, session, and sector at each end of the communication channel and test this using application level commands.

TMW master and slave SCLs support a built in database that will provide reasonable execution for most application layer commands. This built in database should be adequate for this initial testing.

The Triangle MicroWorks, Inc. Communication Protocol Test Harness can be used to facilitate communications testing. The Test Harness is a Windows application that acts as a simple Master or Slave device and can be programmed with an automated test sequence through a scripting capability.

The protocol analyzer output from the Communication Protocol Test Harness allows you to verify the proper operation of the device you are testing. It also allows you to generate a reference protocol analyzer output file that can be used for comparison to later tests to verify only intended modifications have occurred.

The Communication Protocol Test Harness currently supports the following protocol components: IEC 60870-5-101, IEC 60870-5-102, IEC 60870-5-103, IEC 60870-5-104, DNP3 and Modbus.

You may download a full 21-day evaluation version of the Communication Protocol Test Harness (including documentation) from the Triangle MicroWorks, Inc. website at: http://www.TriangleMicroworks.com/downloads.htm.

6.2 Testing Multiple Channels, Sessions, and Sectors

If the user application is designed to support multiple communication channels, sessions, and/or sectors this should be tested once the basic communications are complete. Once again the built in database should be sufficient for this purpose.

6.3 Testing the Database Interface

Depending on the particular implementation, it is generally recommended to implement and test each data type independently. This is easily accomplished by implementing the required database access routines for the desired data type and issuing commands to exercise these routines.

6.4 Testing Event Generation

Event generation can be tested at the same time as the basic database interface or it can be delayed until the database interface is complete.

6.5 Testing Commands

The generation/processing of commands should be tested next. While the execution of some commands will obviously be done in the preceding steps, it is now time to thoroughly test all of the supported commands.

6.6 Regression Testing

Once the device has been thoroughly tested it is highly recommend that all the test scripts be assembled into a complete procedure that can be executed periodically to guarantee continued functionality.

Triangle MicroWorks, Inc. sells Conformance Test scripts that run on the Communication Protocol Test Harness. These scripts perform the conformance tests outlined by the technical committees of each supported protocols.

The scripts can be run in “attended” or “unattended” mode. The attended mode is required to verify complete device performance. The unattended mode skips steps that require specific configuration or values from the device. This mode is useful in automated regression tests.