**Getting started - Nimbus setup**

The Nimbus WebServices use the built-in ActiveX Service and communicates using a single TCP-socket between the WebService and the Nimbus Alarm Server.

Enable Nimbus ActiveX services in the *Nimbus\_Server.ini* file, section *[ActiveX],* parameter *EnableActiveXServer=1*.

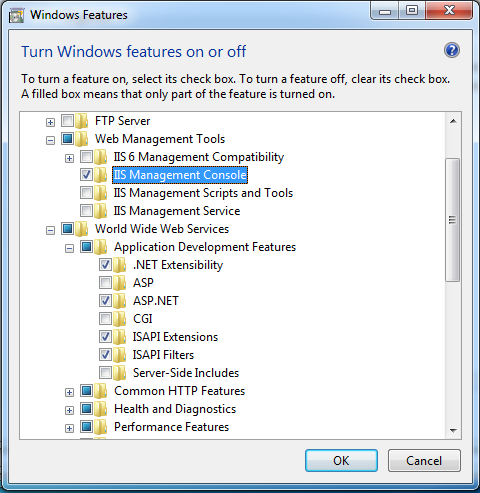
You can open the file explorer in the correct folder by right-clicking the projectpath down below in the Nimbus Explorer window and select 'Open Explorer at this location'.

Don't forget to restart Nimbus Alarm Server when the Nimbus\_Server.ini changes are saved.

You will need Nimbus Alarm Server version 3.00.11 to use the WebServices version 3.0.0.00 and Nimbus Alarm Server version 3.00.14 to use WebServices version 3.0.0.01.

**Getting started - WebService setup**

Install IIS and the *ASP.NET* feature



Copy the *Nimbus Web Service* files to a folder in your website, ex *C:\inetpub\wwwroot\nimbusws*

*.\ NimbusWS.asmx  
.\Global.asax  
. \Web.config  
.\bin\NimbusWS.dll  
.\bin\NimbusWS.pdb*

Edit the *Web.Config* file and change the *ServerAddress* and *ServerPort* to match the settings in *Nimbus\_Server.ini*. It defaults to a local installation at 127.0.0.1.

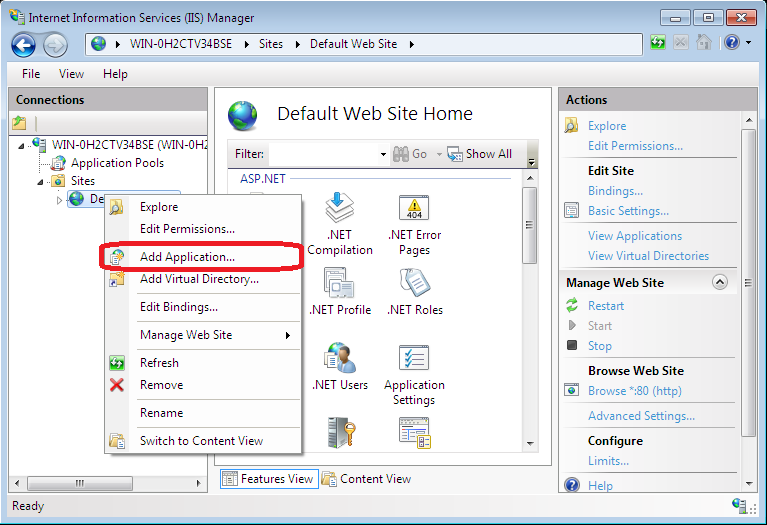
If it is a redundant Nimbus installation, both Nimbus Alarm Server's addresses should be set, ex *ServerAddress="192.168.123.82,192.168.123.83"* and *ServerPort="58658,58658"*

The DNS lookup name could be used instead of the IP

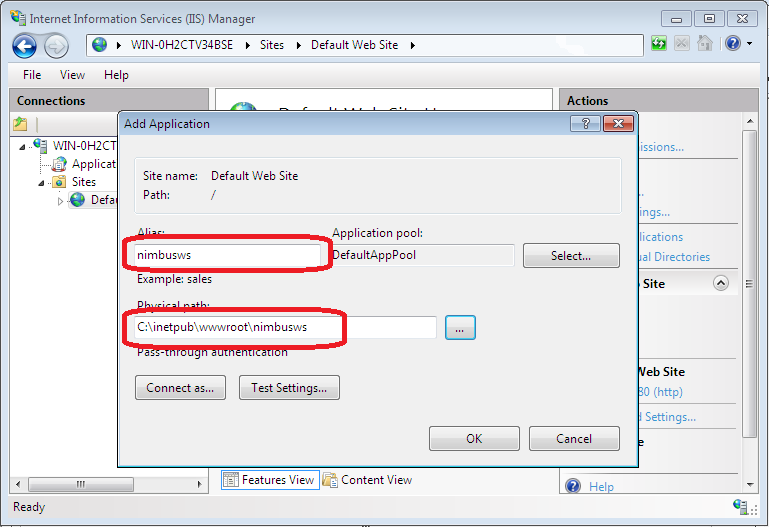
The WebService will automatically connect to the Nimbus Alarm Server in a redundant configuration that is the first to respond and will also automatically try the other server if the connection is broken.

All changes will automatically be replicated to the redundant partner no matter to which server the WebService is connected.

Open IIS Manager

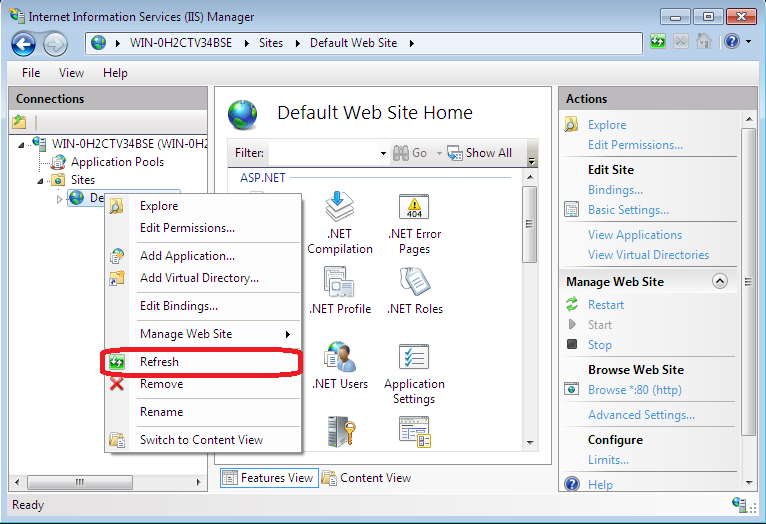


Right-click your web site (i.e. *Default Web Site*), select *Add Application*



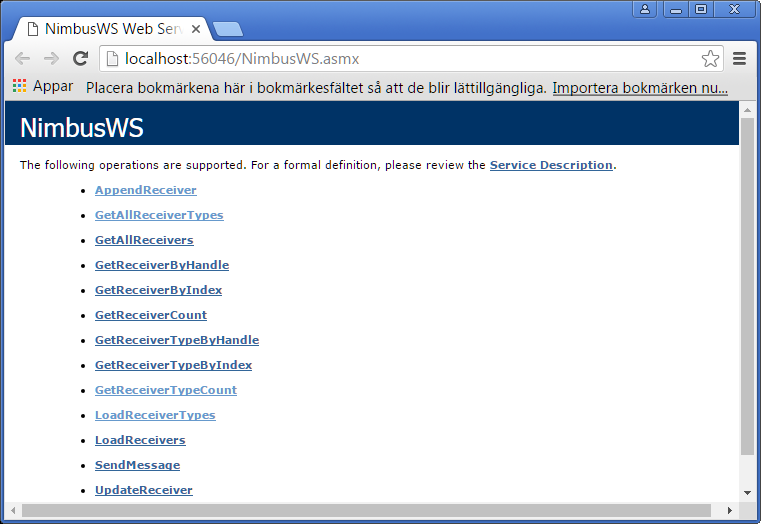
Fill in the info about where you put the files

Right-click your web site (i.e. *Default Web Site*), select *Refresh*



That's it

Open the webservice using *http://localhost/nimbusws/NimbusWS.asmx*



Try to invoke the methods (above example is not complete, methods are described later on)



Example of the *GetAllReceivers* method, which returns an array of class objects containing all receivers



Example of the *GetAllReceiverTypes* method, which returns an array of class objects containing all receivertypes

**Methods**

**IsConnectedToNimbusServer ()**

*Description:*Get the connection status of the Nimbus Alarm Server TCP socket. The status is only ok if the connection is established and all initial data has been retrieved properly

*Return value:*A bool indicating if we are connected and fine (true) or if we are not (false).

**GetEventLog ()**

*Description:*Enumerates the WebService eventlog.

*Return value:*An array of strings (last 100 messages)

**LoadReceivers()**

*Description:*Retrieves all Receivers into the WebService internal collection from Nimbus Alarm Server. The Receivers are automatically retrieved when the WebService is initialized first time, and will automatically be updated from Nimbus Alarm Services if something is changed.

*Return value:*An int containing the number of retrieved Receivers, -1 if we are not connected to Nimbus Alarm Server

**GetReceiverCount ()**

*Description:*Returns the number of Receivers in the WebService collection

*Return value:*An int containing the number of Receivers in the WebService collection, -1 if we are not connected to Nimbus Alarm Server

**GetReceiverByIndex (int Index)**

*Description:*The index value begins with 0 and must be less than *the GetReceiverCount()* value.

*Return value:*A *Receiver*class object

**GetReceiverByHandle (int Handle)**

*Description:*The handle value corresponds to the filename enumeration (ex *Receiver\_1234.dat*).

*Return value:*A *Receiver*class object

**GetReceiverByName (string ReceiverName)**

*Description:*Lookup a receiver using non-case sensitive search in the Receiver collection.

*Return value:*A *Receiver*class object

**GetAllReceivers ()**

*Description:*Enumerates all receivers.

*Return value:*A list of *Receiver*class objects

**EnableOrDisableReceiver(int Handle, string ReceiverName, bool EnableReceiver)**

*Description:*Enable or disable a Receiver.

Only one of the two first parameters are needed. Primary the *Handle* field will be used to find the Receiver. If *Handle* is set to -1 the *ReceiverName* field will be used to find the Receiver.

*Return value:*A bool indicating success (true) or failure (false)

**AddReceiver (cReceiver Receiver)**

*Description:*Adds a new Receiver to the already WebService collection and will it to Nimbus Alarm Server.

The Receiver class should have been filled properly, however the fields *Handle* and*FileDateTimeStamp* are ignored.

If the *ReceiverTypeHandle* field is set to -1 the field *ReceiverTypeName* must be set. If *ReceiverTypeHandle*>= 0 then the *ReceiverTypeName* field is ignored.

*Return value:*An int containing the handle of the newly added receiver. If the receiver could not be added (ex it already exists a receiver with the same name) the handle returned is -1

**UpdateReceiver (cReceiver Receiver)**

*Description:*Update a Receiver in the WebService collection and send it to Nimbus Alarm Server.

The Receiver class object should have been filled properly, however the field *FileDateTimeStamp*is ignored.

If the *ReceiverTypeHandle* field is set to -1 the field *ReceiverTypeName* must be set. If *ReceiverTypeHandle*>= 0 then the *ReceiverTypeName* field is also ignored.

The *Handle* field will be used to find the Receiver. If *Handle* is set to -1 the *ReceiverName* field will be used to find the Receiver. The receiver name may only be changed if *Handle* is valid.

*Return value:*A bool indicating success (true) or failure (false)

**DeleteReceiver (int Handle, string ReceiverName)**

*Description:*Delete a Receiver from the receivers collection in the WebService and send the update to Nimbus Alarm Server.

Only one of the parameters are needed. Primary the *Handle* field will be used to find the Receiver. If *Handle* is set to -1 the *ReceiverName* field will be used to find the Receiver.

*Return value:*A bool indicating success (true) or failure (false)

**CloneReceiver (int Handle, string ReceiverName, string NewReceiverName)**

*Description:*Create a new receiver based upon an old receiver and clone all its properties (beside *Handle* and *ReceiverName*).

Only one of the first two parameters are needed. Primary the *Handle* field will be used to find the Receiver to clone. If *Handle* is set to -1 the *ReceiverName* field will be used to find the Receiver to clone.

*Return value:*An int containing the handle of the newly added receiver. If the receiver could not be added (ex it already exists a receiver with the same name) the handle returned is -1

**LoadProfiles()**

*Description:*Retrieves all Profiles into the WebService internal collection from Nimbus Alarm Server. The Profiles are automatically retrieved when the WebService is initialized first time, and will automatically be updated from Nimbus Alarm Services if something is changed.

*Return value:*An int containing the number of retrieved Profiles, -1 if we are not connected to Nimbus Alarm Server

**GetProfileCount ()**

*Description:*Returns the number of Profiles in the WebService collection

*Return value:*An int containing the number of Profiles in the WebService collection, -1 if we are not connected to Nimbus Alarm Server

**GetProfileByIndex (int Index)**

*Description:*The index value begins with 0 and must be less than *the GetProfileCount()* value.

*Return value:*A *Profile*class object

**GetProfileByHandle (int Handle)**

*Description:*The handle value corresponds to the filename enumeration (ex *Profile\_1234.dat*).

*Return value:*A *Profile*class object

**GetProfileByName (string ProfileName)**

*Description:*Lookup a profile using non-case sensitive search in the Profile collection.

*Return value:*A *Profile*class object

**GetAllProfiles ()**

*Description:*Enumerates all profiles.

*Return value:*A list of *Profile*class objects

**EnableOrDisableProfile (int Handle, string ProfileName, bool EnableProfile)**

*Description:*Enable or disable a Profile.

Only one of the two first parameters are needed. Primary the *Handle* field will be used to find the Profile. If *Handle* is set to -1 the *ProfileName* field will be used to find the Profile.

*Return value:*A bool indicating success (true) or failure (false)

**AddProfile (cProfile Profile)**

*Description:*Adds a new Profile to the WebService collection and sends it to Nimbus Alarm Server.

The Profile class object should have been filled properly, however the fields *Handle* and *FileDateTimeStamp* are ignored.

Dayschedule only needs to be filled if dayschedules are used (UseTimeSchedule = true)

*Return value:*An int containing the handle of the newly added profile. If the profiles could not be added (ex it already exists a profiles with the same name) the handle returned is -1

**UpdateProfile (cProfile Profile)**

*Description:*Update a Profile in the WebService collection and send it to Nimbus Alarm Server.

The Profile class object should have been filled properly, however the field *FileDateTimeStamp* is ignored.

The *Handle* field will be used to find the Profile. If *Handle* is set to -1 the *ProfileName* field will be used to find the Profile. The profile name may only be changed if *Handle* is valid.

*Return value:*A bool indicating success (true) or failure (false)

**DeleteProfile (int Handle, string ProfileName)**

*Description:*Delete a Profile from the profiles collection in the WebService and send the update to Nimbus Alarm Server.

Only one of the parameters are needed. Primary the *Handle* field will be used to find the Profile. If *Handle* is set to -1 the *ProfileName* field will be used to find the Profile.

*Return value:*A bool indicating success (true) or failure (false)

**CloneProfile (int Handle, string ProfileName, string New ProfileName)**

*Description:*Create a new profile based upon an old profile and clone all its properties (beside *Handle* and *ProfileName*).

Only one of the first two parameters are needed. Primary the *Handle* field will be used to find the profile to clone. If *Handle* is set to -1 the *ProfileName* field will be used to find the Profile to clone.

*Return value:*An int containing the handle of the newly added profile. If the profile could not be added (ex it already exists a profile with the same name) the handle returned is -1

**LoadReceiverTypes()**

*Description:*Retrieves the ReceiverTypes from Nimbus Alarm Server into the WebService collection. The ReceiverTypes are automatically retireved when the WebService is initialized first time.

*Return value:*An int containing the number of ReceiverTypes in the collection, -1 if we are not connected to Nimbus Alarm Server

**GetReceiverTypeCount ()**

*Description:*Returns the number of ReceiverTypes in the collection

*Return value:*An int containing the number of ReceiverTypes in the collection, -1 if we are not connected to Nimbus Alarm Server

**GetReceiverTypeByIndex (int Index)**

*Description:*The index value begins with 0 and must be less than *the GetReceiverTypeCount()* value.

*Return value:*A ReceiverType class object

**GetReceiverTypeByHandle (int Handle)**

*Description:*The handle value corresponds to the filename enumeration (ex *ReceiverType\_0021)*

*Return value:*A *ReceiverType*class object

**GetReceiverTypeByName (string ReceiverTypeName)**

*Description:*Lookup a receivertype using non-case sensitive search in the ReceiverType collection.

*Return value:*A *ReceiverType*class object

**GetAllReceiverTypes ()**

*Description:*Enumerates all ReceiverTypes.

*Return value:*A list of *ReceiverType*class objects

**LoadSCADASystems()**

*Description:*Supported SCADA Systems are automatically retrieved from Nimbus Alarm Server when the WebService is initialized.

*Return value:*An int containing the number of SCADASystems in the collection, -1 if we are not connected to Nimbus Alarm Server

**GetSCADASystemCount ()**

*Description:*Returns the number of SCADASystems in the collection

*Return value:*An int containing the number of loaded SCADASystems, -1 if we are not connected to Nimbus Alarm Server

**GetSCADASystemByIndex (int Index)**

*Description:*The index value begins with 0 and must be less than the *GetSCADASystemCount()* value.

*Return value:*A *SCADASystem*class object

**GetSCADASystemByHandle (int Handle)**

*Description:*The handle value for the SCADASystem. The Index value and the *Handle* value are mostly pretty close (or actually the same for some SCADASystems) but they should not be considered to be equal.

*Handle* must be used when a Profile is updated or created (if *ScanSpecificSystems* member andthe *System* listmembers are used).

*Return value:*A *SCADASystem*class object

**GetAllSCADASystems ()**

*Description:*Enumerates all SCADASystems.

*Return value:*A list of *SCADASystem*class objects

**SendMessage (int ReceiverHandle, string ReceiverName, string Message, int DestinationServer)**

*Description:*Place a message in the alarm event queue, destined for the Receiver in *ReceiverHandle* or *ReceiverName*

Only one of the parameters *ReceiverHandle* or *ReceiverName* are needed. Primary the *Handle* field will be used to find the Receiver. If *Handle* is set to -1 the *ReceiverName* field will be used to find the Receiver.

The *DestinationServer* parameter is only used in a redundant configuration:  
  
*DestinationServer* = 0 -> Send from the primary Nimbus Server only  
*DestinationServer* = 1 -> Send from the secondary Nimbus Server only  
*DestinationServer* = 2 -> Send from the active Nimbus Server only  
*DestinationServer* = 3 -> Send from both Nimbus Servers  
  
In a single server configuration, the parameter should be set to 0

*(As for now (June 2017 Nimbus release 3.00.11) the DestinationServer has no function)*

*Return value:*An int indicating the message id used internally in Nimbus Alarm Server or -1 if it failed. This message id (event id) also is used in the Nimbus log files. The return value only reflects if the message was successfully placed in the queue or not, it does not indicate the result of the sending process.

**QueueAlarm (cAlarmEvent AlarmEvent, int DestinationServer)**

*Description:*Place an alarm event in the alarm event queue. The AlarmEvent will be treated as an alarm arriving from the SCADA system specified in the *SCADASystemHandle* parameter.

The SCADA System handles are sort of static and will probably never be changed, for example *Generic File* will always have handle *36*.

The *DestinationServer* parameter is only used in a redundant configuration, see *SendMessage* for explanation.

*(As for now (November 2017 Nimbus release 3.00.14) the DestinationServer has no function)*

*Return value:*A bool indicating success (true) or failure (false). The return value only reflects if the alarm was successfully placed in the queue or not. It does not indicate the result of the sending process.

**AbortAll (int DestinationServer)**

*Description:*Abort all queued events waiting to be sent. In a redundant system the abort function is performed only on the primary server.

The *DestinationServer* parameter is only used in a redundant configuration, see *SendMessage* for explanation.

*(As for now (November 2017 Nimbus release 3.00.14) the DestinationServer has no function)*

*Return value:*A bool indicating success (true) or failure (false).

**SubscribeForNimbusAlarmEvents(bool Enable)**

*Description:*Nimbus could also send incoming alarm events to the webservice. They will be queued and the method *GetNimbusAlarmEvents*() must be used cyclically to retreive new events.

Alarm events sent to Nimbus using the *QueueAlarm()* method will not be sent back to the same WebService client. IIS will from the WebService perspective be one client, this could lead to alarm events sent using an IIS client connection not be sent back hence not visible to other IIS clients. Using this method to stop subscribe will then affect all IIS clients.

The default behaviour is set by the Web.Config-parameter *SubscribeForAlarmEvents*. The queue size may be changed with the *MaxNimbusAlarmEvents* parameter (defaults to 1000 events)

*Return value:*A bool indicating success (true) or failure (false).

**GetLastNimbusAlarmEventIndex ()**

*Description:*Gets the last alarm event index. If no alarm events have been sent from Nimbus, the index equals to -1.

The last index is always returned by the *GetNimbusAlarmEvents()* method, but calling this method first time with index -1 will return all alarm events queued since the WebService connected to Nimbus (upto *MaxNimbusAlarmEvents*events). To avoid this the*GetLastNimbusAlarmEventIndex()* methodcould be used to retrieve future alarm events.

*Return value:*An int with the last alarm index.

**GetNimbusAlarmEvents (ref int LastEventIndex)**

*Description:*Get the alarm events that occurred since *LastEventIndex*. The variable *LastEventIndex* will automatically be updated and should be used when calling the *GetNimbusAlarmEvents* method next time, or the WebService will return the same list again.

This method should be called cyclically.

*Return value:*A list of *NimbusAlarmEvent* class objects.

**GetIniParameter (string Section, string Parameter, string Default, int DestinationServer)**

*Description:*Gets an INI parameter from the Nimbus\_Server.ini file.

The *DestinationServer* parameter is only used in a redundant configuration, see *SendMessage* for explanation.

*(As for now (January 2019 Nimbus release 3.00.17 B2) the DestinationServer has no function)*

*Return value:*The parameter value or the default value if the parameter does not exist.  
If the function fails, a null value is returned.

**GetRunningApp (int ApplicationIndex, int DestinationServer)**

*Description:*Gets info about running apps and active function in Nimbus, as follows:

*0 - Nimbus Alarm Server is running  
1 - Nimbus Explorer is running  
2 - (not used)  
3 - Nimbus Alarm Receiver is running  
4 - Nimbus Alarm Server is running as a service  
5 - Nimbus Alarm Server alarm sending is blocked  
6 - Nimbus Alarm Server is running in demo mode  
7 - (not used)  
8 - Nimbus is Primary server  
9 - Nimbus is Secondary server  
10 - Nimbus is the Active server*

The *DestinationServer* parameter is only used in a redundant configuration, see *SendMessage* for explanation.

*(As for now (January 2019 Nimbus release 3.00.17 B2) the DestinationServer has no function)*

*Return value:*A bool indicating if the app and/or function is active.

**GetRunningApps (int DestinationServer)**

*Description:*Gets info about running apps and active function in Nimbus

The *DestinationServer* parameter is only used in a redundant configuration, see *SendMessage* for explanation.

*(As for now (January 2019 Nimbus release 3.00.17 B2) the DestinationServer has no function)*

*Return value:*A bool array (32 elements) indicating if the apps and/or functions are active, see list in *GetRunningApp()* or *null* if the function failed.

**Structs and classes**

**ReceiverType**

publicclasscReceiverType

{

publicstring ReceiverTypeName;

publicint Handle;

publicstring Algorithm;// Type of algoithm to be used when sending

publicint Limit;// Max number of chars in messages

publicstring SettingFileName;

publicstring TemplateFileName;

publicstring Country;

}

**Receiver**

publicclasscReceiver

{

publicconstint MAX\_RECEIVERSETTINGS = 4;

publicstring ReceiverName;

publicstring BackupReceiver;

publicint Handle;

publicbool Enabled;

publicDateTime FileDateTimeStamp;// Timestamp of last change, read only

publicint KeepAliveInterval;

publicint ReceiverTypeHandle

publicstring ReceiverTypeName;

publicstring[] Setting; // Setting fields depends of the receiver type

}

**Profile**

publicclasscProfile

{

publicconstint MAX\_CONDITIONS = 6;

publicconstint MAX\_SYSTEMTYPES = 64;

publicstring ProfileName;

publicint Handle;

publicbool Enabled;

publicbool IncludeInactiveAlarms;

publicbool IncludeActiveAlarms;

publicbool IncludeAckedAlarms;

// If true also timeschedule is used as filter criteria

publicbool UseTimeSchedule;

// List of parallell receivers

publicstring[] ParallellReceivers;

// List of sequential receivers

publiccSequentialReceiver[] SequentialReceivers;

// The filter criterias (Tag, Area, Name etc)

// Max 6 items (usually named t0..t5), unallocated items are set to ’\*’

publicstring[] Conditions;

publicbool TimeSchedule\_ViewDayType;

publicbool Calendar\_ViewTimeAxis;

publicbool UseExternalTag;

publicstring ExternalTagName;

publicbool IgnoreExternalTagInOtherProfiles;

publicbool UseExternalFile;

publicstring ExternalFileName;

publicstring ExternalFileText;

publicbool UseAutoBlock;

publicint AutoBlockTresholdEvents;

publicint AutoBlockTresholdTime;

publicint AutoBlockTime;

// true if this profile should be applied to specific SCADA systems

// If true the specific SCADA system handles are specified in the System

// list

publicbool ScanSpecificSystems;

// List of SCADA System handles this profile will be used for if

// ScanSpecificSystems is true. Order of the handles are irrelevant

publicList<int> System;

// 0 – Throw alarm (standard behaviour)

// 1 – Delay sending until on duty time is passed

// 2 – As 1 but throw queued event if it becomes inactive

// 3 – As 1 but throw queued event if it becomes acked

// 4 – As 1 but throw queued event if it becomes inactive or acked

publicint OffDutyAlarmBehaviour;

// Last profile update timestamp (read only)

publicDateTime FileDateTimeStamp;

// Max 9 days may be defined (Weekdays + Holiday evening

// + Holiday)

publicList<cDaySchedule> DaySchedule;

// If calender should be used or not (global calender cannot be

// changed using webservices

publicbool UseCalendar;

}

publicstructcSequentialReceiver

{

// Receiver name

publicstring ReceiverName;

// true if this receiver is needed to manually confirm alarm

// reception

publicbool RequestConfirmation;

}

publicstructcDaySchedule

{

// If days schedule is to be used. The schedule 0..6 (mo..su)

// in DaySchedule are always enabled

publicbool InUse;

publicbool Grouped;

// Start/stop periods

publicList<cOnOff>Periods;

}

publicstructcOnOff

{

// Start/stop times in DaySchedule[]

publicint StartTime;// 0-1440 equals 00:00..24:00 (minutes since midnigth)

publicint StopTime;// 0-1440 equals 00:00..24:00 (minutes since midnigth)

}

**SCADASystem**

publicclasscSCADASystem

{

publicstring Name;

publicint Handle;

publicstring Description;

}

**AlarmEvent**

publicstructcAlarmEvent

{

// 0 - Inactive (Normal), 1 - Active (Alarm), 2 – Acked

publicint EventType;

// Alarm date in format YYYY-MM-DD, if omitted it is set to now

publicstring Date;

// Alarm date in format HH:NN:SS, if omitted it is set to now

publicstring Time;

publicstring Tag; // [t0]

publicstring Area; // [t1]

publicstring Category; // [t2]

publicstring Name; // [t3]

publicstring Description; // [t4]

publicstring Status; // [t5]

publicint SCADASystemHandle; // Simulated SCADA system

}

**NimbusAlarmEvent**

publicclasscNimbusAlarmEvent

{

publicenumAlarmEventType

{

InActive = 0,

Active = 1,

Acked = 2

}

publicAlarmEventType alarmEventType;

publicint EventId;

publicDateTime TimeStampFromPC;

publicDateTime TimeStampFromSCADA;

publicDateTime TimeToSend;

publicint System;

publicstring Tag;

publicstring Area;

publicstring Category;

publicstring Name;

publicstring Description;

publicstring Status;

}