Cascadas Protocol Specifications

Table of Contents

Authors: Juan José Denis Corrales, Juan Miguel Taboada Godoy, Michael Griffin

Version: 0.9.2 of 2009-12-16

---

1. Introduction

The Cascadas protocol is designed for communicating between an HMI client and a data server. Other applications are also possible. The protocol is simple, and concentrates on supporting the high level features required in typical HMI applications.

The protocol allows for easy implementation of platform independent web based HMI systems as well as traditional HMI systems.

The Cascadas protocol supports the following features:

- Tag based addressing.
- Read and write data.
- Events.
- Alarms.
- Alarm history.
- Error detection and reporting.

This document describes the complete protocol specifications, together with application guidelines and examples.

---

2. Functional Description

2.1 Area of Application

The Cascadas protocol is a simple communications protocol that is intended for applications in HMI (Human Machine Interface) systems as well as in other applications such as MES (Manufacturing Execution System) integration. It is particularly well suited for web based HMI applications transported over HTTP.

For web based HMI applications, the client would be a computer running a web browser that uses a web page (HTML, XHTML, SVG, Javascript) or Flash (swf) file. These may include animated graphs related to the dynamic data stored in a data server. For non-web based HMI applications, the client can be a traditional PC application which uses the protocol to communicate with the data server.

The server can be any program (a web server, or a stand alone program) running in any device that holds data values (pressure, switches on and off, alarms, etc.) retrieved by various means from the field. The data server would communicate with the client using the Cascadas format.

The server would be expected to have a current copy of the data which is to be displayed on the HMI. That data would be stored in a data table (array of memory), database, or some other data structure. If the web server is part of the controller being monitored, it may access the controller's data structures directly.

A single server may support multiple clients simultaneously. There is nothing inherent to the protocol which limits the number of clients which may be supported. Physical limits will be dictated by server software load, but in most practical applications this limit would never be reached.

For MES integration applications, the MES system can be the client or server, and can use the basic read/write functionality of the protocol to exchange data with a compatible source.

The rest of this document will concentrate on HMI applications, but the same features may be used for non-HMI applications.

2.2 Typical System Operation

The HMI that is displayed to the user would be a web page that is created for the specific application and updated on a regular basis using information from the server. A web based HMI system would typically operate as follows:

1. The web browser requests the web page from the server. The name of the web page requested (the URL) determines what HMI is displayed to the user.
2. The server receives the request, and sends the appropriate web page.
3. If the web page consists of multiple files (graphics files, external JavaScript files, logos, etc.), the additional files are automatically delivered as the web browser requests them.
4. The web browser executes the client JavaScript (or Flash, or other scripting) embedded in the page.
5. The client scripts send an HTTP request back to the original server using the protocol described in this document. The client request may include data which is to be written to the server's storage (data table, database, etc.).
6. The server replies to the request with an HTTP message using the protocol described in this document.
7. The client updates its web page using data from the server's response.
8. The client waits some defined period of time (e.g. 1 second).
9. The request/response process repeats indefinitely.

2.2.1 Client Responsibilities

Under this scenario, the client is responsible for:

- Initiating communications with the server.
- Polling the server on a regular basis for new data.
- Displaying the HMI screens and data to the user.
- Handling any errors in communications with the server.

2.2.2 Server Responsibilities

The server is responsible for:

- Authenticating the client and restricting access (if required). Authentication and access control are not defined by this protocol.
- Sending the requested HMI web page to the client.
- Responding to client requests.
- Checking any information received from the client to ensure the client is making a valid request.
- Converting values in the protocol messages to whatever internal format is used by the server. This may include re-scaling and converting types.
- Storing data for use in the HMI system. This may be in a database, data table (array), or some other form of storage.
- Maintaining message buffers for events and alarms.
- Communicating with other devices to obtain the data required by the HMI client, and to communicate user inputs from the client to whatever other parts of the system may need to make use of it.

2.3 Protocol Transport Formats

The protocol is transport agnostic. Current implementations use HTTP, but direct socket and other transports are possible.

For HTTP transport, requests from the client to the server are embedded as JSON in the headers as "Cascadas:". Responses from the server to the client are returned as JSON documents following the header.

2.4 Data Encoding (JSON)

The Cascadas protocol is based on JSON (JavaScript Object Notation), which is a popular lightweight protocol that is readily supported in Internet applications. JSON has a number of advantages, including being simple, consistent, well defined, and easily human readable. It combines this with being reasonably close in syntax to many common programming languages making the translation to a from usable form relatively easy and straight forward. More information about JSON may be found at www.json.org

A JSON message corresponds closely to a JavaScript "object". A description of JavaScript is outside of the scope of this document, but the following explanation should be sufficient to understand how JSON is used here.

A JavaScript **object** consists of a series of key/value pairs. The "key" will be a string which is used to refer to the data value by name. The "value" is a number, string, array of data, or object. The key is separated from the value by a colon ":". Each key/value pair is separated from other key/value pairs by a comma ",". For example, "key" : value.

The beginning of an object is indicated by an opening brace "{", and the end of an object is indicated by a closing brace "}". Objects may be nested inside other objects.

For example, {"key1" : 123} and also {"key1" : {"key2" : 421, "key3" : 930}}

An array is simply a series of values separated by commas and enclosed within square brackets.

For example [123, 4.78, "abc"]

Example of a JSON message:

```
{"key1":123, "somekey":"value", "arrayexample":[1, 2, 3]}
```

JSON is designed to be closely related to JavaScript because JavaScript is the scripting language used in web browsers. Other programming languages use a similar syntax for data structures. In these other languages, an "object" has a somewhat different meaning than how the term is used in JavaScript, and the name "dictionary" or "associative array" is used in place of what JavaScript refers to as an "object". This document will sometimes use the term "dictionary" to help clarify the meaning of "object".

2.5 Security Models

The protocol does not provide its own security, but rather relies on the transport layer to provide security using means which are appropriate to the application.

Typical security models which may be applied are:

Command data	*Log-in*	*Comm*	*Security*
Anonymous	None	In clear	None. Similar to typical industrial protocols. Only suitable where authorization and security are not required.
Basic authentication	In clear	In clear	Minimal. Helps prevent caused by unintentionally logging into the wrong server, but will not protect against a deliberate attack. Does not protect passwords from being "sniffed" by someone monitoring the network connection.
Secure log-in	Encrypted	In clear	Protects passwords, but does not prevent a malicious party from sending forged commands.

Full encryption	Encrypted	Encrypted	Protects passwords and protocol commands, but may not fully protect against a "replay" attack.

Anonymous access is similar to how typical industrial HMI and I/O communications are usually implemented. They offer no security and so are applicable only where none is required or where security is provided by some other means (e.g. tunnelling the communications over a VPN). Since many industrial HMI applications do not operate in an environment where security is required, this is the model which most users will be familiar with.

Basic authentication requires the user to log in with a user name and password. The password is sent in plain (not encrypted). This provides a degree of protection against error by helping prevent someone from logging into the wrong server. It also helps prevent casual access by unauthorized but non-malicious users. However, since the password is sent in plain, a malicious person who was able to monitor traffic on the network might be able to read the password and use this to gain access. Also, since the message traffic itself is not encrypted, a malicious person who had access to the network might read the messages and observe confidential information, or even forge commands to send to the server. This level of security though is greater than what is used in most HMI applications, and may be adequate in many cases.

Secure log-in requires the user to log in over an encrypted channel (typically HTTPS). Subsequent communications after log-in are sent in clear. The advantage over basic authentication is that the passwords are protected. The disadvantage is that it is usually necessary to install a signed security certificate in each web browser being used as an HMI.

Full encryption uses encryption for both the log-in and all subsequent communications. This has the advantages of protecting the protocol messages from being monitored. However, it adds considerable overhead at both the client and the server for encryption. It may not protect against "replay" attacks, where a malicious third party records messages and "replays" them again later.

3.0 Protocol Elements and Definitions

The protocol contains the following elements:

- **Message** - A message is a collection of **commands**, enclosed in braces {}, and separated by commas.
- **Commands** - A command is a pair of string (called **command key**) and value (called **command data**) that define an action or a flow of information.
- **Command keys** - Command keys are text strings which identify the individual major message elements. Examples of command keys are: "read", "write", etc.
- **Command data** - The values may be a basic type (e.g. integer or string), or it may be a list of values enclosed in brackets [] (javascript *array*, python *list*) or a collection of string/value pairs enclosed in braces {} (javascript *object*, python *dictionary*). It also may be a compound type containing nested Data tags with multiple basic types. If the data is present in string/value pairs, the string is called a *tag*.
- **Data tags** - A tag is a string which represents an individual data value within **command data** (a data table memory location, database entry, or other storage location). All data is addressed by means of tags.
- **Data values** - A data value is the data addressed by means of tags.

Let's see an example:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":0,
"write" : {"SOL1" : 0, "SOL2" : 1, "Pump1Speed" : 2250},
}
```

This message has three commands: the first one identifies the client, the second one identifies the message, and the third writes data on the server. The command keys are "id", "msgid" and "write".

"SOL1", "SOL2", and "Pump1Speed" are tags, representing positions in a data base.

3.1 Data Types

3.1.1 Basic Data Types

Data may be sent as part of a request or response. The following data types are defined:

Type Name	*Type Description*	*Range*
Boolean	Boolean	0 = False, 1 = True
Integer	Signed decimal integer.	The data range is implementation dependent.
Floating point	Numeric floating point	The data range is implementation dependent.
Time	Time in seconds since the 1st of January 1970 UTC (Unix Epoch).	
String	String	The maximum length is 256 characters.

Time may be represented as an integer or floating point number. The precision of the time value is platform dependent.

3.1.2 Compound Data Types

In addition to the basic data types, there are also compound data types. Compound data types are collections of basic data types and / or commands. These include the following:

- **List** - This is a list or array of tags or objects. Lists may contain tags, other lists, or objects.
- **Object** - This is a collection of key/value pairs. The values may be basic data types, lists, or other objects. A complete Cascadas message is a type of nested object.

Example of a list:

```
["PB1", "PB2", "PB3"]
```

Example of an object:

```
{"SOL1" : 0, "SOL2" : 1, "Pump1Speed" : 2250}
```

3.1.3 Type Conversion

When a client or server receives a data item which is not of the expected type, it is to attempt to convert the data to the expected type according to the following rules:

- Integers are converted to floating point numbers of the same magnitude.
- Floating point numbers are converted to integer by truncating.
- Numbers are converted to boolean by first converting to integer, and then testing to see if they equal 0 or 1.

Type conversion is to be conducted on a "best effort" basis. For some floating point values it is not possible to represent them accurately in binary form. In these cases, converting a floating point number which is very close to the next higher integer value may result in the number being rounded up instead of truncated.

3.2 Data Tags

There are six different types of data tags:

- **Message syntax tags** - Those tags used to define the extent of the command.
- **Address tags** - A user defined tag, such as an input, output, or other implementation specific storage location. Used in read or write commands.
- **Reserved tags** - Used in read or write commands to query specific data from the server. These tags may not be used by the client for user defined tags.
- **Event tags** - An event tag is used to identify a *type* of event.
- **Alarm tags** - Identifies the occurrence of a singular alarm.
- **Zone tags** - Identifies a zone in which alarms or events occur.

Tag must observe these format rules:

- **Tag Length** - Tags are to be no more that 100 characters long. Some client platforms may have an implementation specific limit which is less than this.
- **Valid Characters** - Tags may contain the letters 'a' to 'z', 'A' to 'Z', the digits '0' to '9', and '_'. Tags may not start with a number.
- **Character Encoding Set** - UTF-8.

3.2.1 Reserved Tags

The following reserved tags may be sent by the client to the server in a read request. The server must respond with a valid reply. The response is returned as with any normal read request.

These tags may not be used by the client for user defined tags.

Tag Name	*Type*	*Description*
timeutc	Time	Time in UTC.
timelocal	Time	Local time at a specified location. The location chosen as local is implementation dependent and may be different for each client.
clientversion	String	The current version of the copy of the client present at the *server*. This is to allow web clients to determine if a different version of the client is present at the server. This is intended to allow automatic upgrades or version roll-backs. How the client makes use of this information is implementation dependent.
protocolversion	String	The current version of the protocol supported.

An implementation may also define additional reserved tags according to need. These may be read-only, write-only, or read-write.

The server is responsible for mapping of address tags to whatever internal representation the server uses for storage locations. The user may define any tag which is not a reserved tag.

3.3 Ordering of Elements

Command Fields may appear in any order in a message. The protocol does not guarantee the order in which command fields will appear will be consistent from message to message.

Data tags may appear in any order in a command field value. The protocol does not guarantee the order in which Data tags will appear will be consistent from message to message or even within an individual message.

Other command fields are optional and are only required to be included if they contain data. If a command field normally contains compound data, the command field name may appear without any associated data by sending an empty object ("{}") or array ("[]") in place of the data (the empty response must be compatible with the expected type). If a command field only expects a simple data type however, then

empty objects or arrays may not be used as placeholders.

---

4. Protocol Control Commands

Certain basic commands must be present in all messages and are used to control the operation of the protocol. These commands are "id", "msgid" and "stat".

- **id** - The "id" command is a string used to identify the client and the server. The client sends its id with each request, and the server sends it own id in the response. "id" must appear in all requests and responses.
- **msgid** - The "msgid" command is used to identify consecutive messages. It consists of an integer which the client would normally increment for each consecutive message. The msgid is generated by the client, and echoed by the server in its response. "msgid" must appear in all requests and responses.
- **stat** - The server returns a status code with the "stat" command to report its status. "stat" must appear in all responses. It does not appear in requests.

"id", and "msgid" are **mandatory** commands and must appear in all requests and responses. "stat" is **mandatory** and must appear in all responses.

Example request:

```
{
   "id":"HMI 9876 from Water Pressure INC.",
   "msgid":1234,
}
```

Example response:

```
{
   "id": "HMIServer demo server",
   "msgid": 0,
   "stat": "ok",
}
```

---

5. Time Stamp

- **timestamp** - The "timestamp" field is returned by the server to indicate the current time stamp.

"timestamp" is returned by the server to indicate the current time. This is always GMT (UTC), and if local time is desired it must be converted by the client.

"timestamp" differs from the protocol control commands (id, msgid, stat) in that it is optional and is not required to appear in the response.

Example request:

```
{
   "id":"HMI 9876 from Water Pressure INC.",
   "msgid":0,
}
```

Example response:

```
{
"id": "HMIServer demo server",
"msgid": 0,
"stat": "ok",
"timestamp": 1238156675.0,
}
```

---

6. Basic Read / Write Operations

The most basic operations are reading or writing real time data using the "read" and "write" commands. Errors are reported by the server in the readerr and writeerr commands.

- **read** - The "read" command requests data from the server. In a request, the data is a list of address tags for which the client wishes to receive the current data values. In a response, the data is an object of key/value pairs containing the tags and their corresponding data values.
- **write** - The "write" command attempts to write data to the server. The data is an object containing key/value pairs. The request contains values which the client wishes to write to the server. The write command is not present in the response.
- **readerr** - The "readerr" command is used to report errors from the "read" command. This is an object of key/value pairs containing the tags used in the previous "read" request and their corresponding error codes. Only tags for which an error was encountered are returned. If there were no errors, the response is empty or not present.
- **writeerr** - The "writeerr command is used to report errors from the "write" command. This is an object of key/value pairs containing the tags used in the previous "write" request and their corresponding error codes. Only tags for which an error was encountered are returned. If there were no errors, the response is empty or not present.

The following is an example of a typical request:

```
{
   "id":"HMI 9876 from Water Pressure INC.",
   "msgid":0,
   "read":["PL1", "PL2", "PL3", "PL4", "Tank1Level"],
   "write":{"PB1":1}
}
```

The following is an example of a typical response with no errors:

```
{
"id": "HMIServer demo server",
"msgid": 0,
"stat": "ok",
"timestamp": 1238156675.0,
"read": {"PumpSpeedActual": 0,
         "PL4": 0,
         "PL3": 1,
         "PL2": 1,
         "PL1": 0,
         "Tank1Level": 40}
}
```

The following is an example of a typical request with invalid read and write commands:

```
{
   "id":"HMI 9876 from Water Pressure INC.",
   "msgid":10,
   "read":["PB1", "TankLevel", "PL2"],
   "write":{"SOL1":1, "Pump1Speed" : 100, "PB1":1}
}
```

The following is an example of a typical response with errors. In this example, "PB1", "TankLevel", "SOL1", and "Pump1Speed" encountered errors, while "PL2" and "PB1" completed successfully:

```
{
"id": "HMIServer demo server",
"msgid": 0,
"stat": "ok",
"timestamp": 1238156675.0,
"read": {"PL2": 1},
"readerr" : {"PB1" : "tagnotfound", "TankLevel" : "servererror"},
"writeerr" : {"SOL1" : "tagnotfound", "Pump1Speed" : "servererror"}
}
```

Note that the server doesn't respond to a successful write operation The result of the operation may take an undefined amount of time to be reflected in the control system. The length of time this takes will be determined by the design of the overall system, and not by the protocol.

---

7. Read with Notify on Change

The "readnoc" command allows read operations to take place using "notify on change". NOC allows the server to send only that read data which has changed since the last response and to not send data which has not changed. In addition, the client needs to send the tag list only once. This saves communications bandwidth and results in less workload for the HMI client at the expense of somewhat more complexity and workload for the server.

- **readnoc** - The read with NOC ("readnoc") operates in a manner similar to the normal "read" command.

The operation is as follows:

1. To start using "readnoc", the client must send a request containing the list of tag addresses which are to be read. This list is formatted in a manner identical to that used for the normal "read" command.

2. The server will reply with an object (dictionary) containing all the tags and values. This response is formatted in a manner identical to that used for the "read" command.

3. As well as responding to the request, the server will "remember" all the tags as well as the values associated with them. This will be stored in a manner which allows the tags and values to be associated with the client making the request.

4. When the client makes a subsequent request, it will send an empty list (array) with the "readnoc" command.

5. When the server receives the empty tag list, it will respond with only those tags whose values have changed since the last response.

6. If the client sends a new tag list, the existing tag list will be discarded and the new tag list used to process requests.

The server must track the NOC list for each client individually. The server cannot share NOC lists between multiple clients even if the clients have identical lists. This is because reading using NOC modifies the "remembered" values, which means that an NOC read operation will affect the response given for the next NOC read operation.

The client using NOC must track all requests and ensure that a valid response is received. Since only changed data is sent, the loss of a single message will not automatically correct itself on the next polling cycle. If the client does not receive a response to an NOC request, it must send the tag list again to start the NOC cycle over again from the beginning.

Normal errors are returned in "readerr".

If the client sends an empty NOC list and there is no NOC list present in the server, the server will return a status code of "noclistempty". This will not be reported as an error in "readerr". The client should handle this by sending the NOC list to the server again. If the server believes that an unrecoverable NOC error has occurred, it should discard the NOC list for that client and send a status of "noclistempty" to the client.

Examples

Request:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":0,
"readnoc" : ["PB1", "PB2", "LS1", "SS5", "TankLevel"]
}
```

Response:

```
{
"id": "HMIServer demo server",
"msgid": 0,
"stat": "ok",
"timestamp": 1238156675.0,
"readnoc" : {"PB1" : 1, "PB2" : 0, "LS1" : 0, "SS5" : 1, "TankLevel" : 50.67}
}
```

Example - Subsequent polling cycle when no data has changed:

Request:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":1,
"readnoc" : []
}
```

Response:

```
{
"id": "HMIServer demo server",
"msgid": 1,
"stat": "ok",
"timestamp": 1238156676.0,
"readnoc" : {}
}
```

Example - Subsequent polling cycle when "LS1" (and no other tag) has changed:

Request:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":100,
"readnoc" : []
```

```
}
```

Response:

```
{
"id": "HMIServer demo server",
"msgid": 100,
"stat": "ok",
"timestamp": 1238156776.0,
 "readnoc" : {"LS1" : 1}
}
```

---

8. Read and Write Access Verification

On starting a client may wish to first check to see if it has permission to read and write all necessary tags and that the data types are as expected before entering into normal operation. If the client does not check access to all address tags, it may find later that due to a programmer error or an incorrect configuration, it doesn't have write access to a tag and the operator is unable to activate an essential device.

- **readable** - The "readnoc" command tests read access.
- **writable** - The "writable" command tests write access.

A client may check any tag without causing any data changes or other side effects by using the "readable" and "writable" commands. These check the status and type of read and write tags, but do not perform any read or write operations on those tags.

Once the client has confirmed that everything is OK, it can start normal operation.

We have found these commands very useful when configuring the client.

The following is an example of a typical request:

```
{
"id" : "HMI 9876 from Water Pressure INC.",
"msgid" : 12345,
"readable" : {"PB1" , "TankLevel" },
"writable" : {"SOL1" , "Pump1Speed"}
}
```

The following is an example of a typical response with no errors:

```
{
"id"    : "Scada 10934 from Water Pressure INC.",
"msgid"  : 12347,
"timestamp": 129873294,
"stat" : "ok",
"readable" : {"PB1" : "boolean", "TankLevel" :"integer"},
"writable" : {"SOL1" : "boolean", "Pump1Speed" :"integer"}
}
```

If there are no errors, the server may omit the "readable" and "writable" response.

---

9. Alarms and Events

Alarms and events differ from ordinary address tags in that there is additional state information attached to them, and that they are only returned in a response when there is new information that is of interest to the operator.

- **events** - The "events" command retrieves a selected group and sequence of events.
- **alarms** - The "alarms" command reads the current state of a selected group of alarms.
- **alarmack** - The "alarmack" command acknowledges selected alarms.
- **alarmhistory** - The "alarmhistory" command retrieves a selected group and sequence of alarm history.

9.1 Alarms

Alarms are conditions which must be acknowledged by the operator. Alarms represent the current state of a condition which is being monitored. When a client receives a new alarm response from the server, the new state of the alarms completely replaces any existing state. When an alarm becomes inactive, it will no longer be present in the response.

A particular alarm is identified by its tag. For example, the alarm tag "tank1overflow" may identify an event described as "Tank #1 has overflowed". However, an alarm is requested from the server by requesting the "zone" of which that alarm is a member. Alarms are never requested by their alarm tags.

Alarm tags are used in alarm messages, alarm acknowledgements, and alarm history messages. Each alarm tag would normally be associated with a memory address in the server, or with a logical result which is a combination of two or more memory addresses in the server.

9.1.1 Alarm States

An alarm is considered to be "active" when it is in any state other than "inactive". Active alarms are normally displayed to the operator. The "ackok" state is a transitional state that may be required to correctly process transitions between "ok" and "inactive". Normally, the "ackok" state should not result in a visible message for the operator.

The valid alarm states are:

#	*State*	*Description*
1	"alarm"	The alarm condition is true, and the alarm has not been acknowledged.
2	"ackalarm"	The alarm condition is true, and the alarm has been acknowledged.
3	"ok"	The alarm condition is false, but the alarm has not been acknowledged yet.
4	"ackok"	The alarm condition is false, and the alarm has been acknowledged.
5	"inactive"	The alarm condition is false, and the alarm has been acknowledged.

9.1.2 Alarm Data

When the server returns an alarm response which contains active alarms, the response will include the following information:

- **Tag name** - This is the alarm tag name. This is used to identify the particular alarm.
- **state** - This is the current state of the alarm.
- **count** - This is the number of times the alarm has gone in and out of the alarm state while active in this cycle.
- **time** - This is the time at which the alarm became active. Once applied, this time stamp remains unchanged as long as this alarm remains active.
- **timeok** - This is the time at which the condition the alarm condition become false. If the alarm condition is true, this value is 0 (or 0.0). If the alarm condition changes from false to true while the

alarm is active, the value is reset to 0.

Examples

Request, the client requests and active alarms from zone1, zone2, or zone3:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":829,
"alarms":["zone1", "zone2", "zone3"],
}
```

The server responds with any active alarms in those zones:

```
{
"id": "HMIServer demo server",
"stat": "ok",
"msgid": 829,
"timestamp": 1238157917.0,
"alarms": {"PB2Alarm": {"count": 1, "state": "alarm", "timeok": 0.0,
                        "time": 1238138641.5069301},
           "PB3Alarm": {"count": 2, "state": "ackalarm", "timeok": 0.0,
                        "time": 1238138614.233047},
           "PB1Alarm": {"count": 2, "state": "ok",
                        "timeok":1238139638.2538631, "time":
                        1238138638.3152909}}
}
```

The client asks for any active alarms in zone4 and zone5:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":830,
"alarms":["zone4", "zone5"],
}
```

The server returns a response indicating there are no active alarms in those zones:

```
{
"id": "HMIServer demo server",
"msgid": 830,
"stat": "ok",
"timestamp": 1238157919.0,
"alarms": {}
}
```

9.2 Alarm Acknowledge

An alarm acknowledgement is an action by the operator indicating that the operator is aware of the alarm. When an alarm is acknowledged, the alarm will change state as outlined above. The HMI client will normally display acknowledged alarms in a different manner from alarms which have not been acknowledged. This allows the operator to more easily see when new alarms appear.

A client acknowledges an alarm by sending a copy of the alarm tag to the server. The server will process the alarm tag and update the state of the active alarm which corresponds to the alarm tag.

If the server receives an acknowledgement for an alarm which does not exist, is inactive, or for which the client is not authorized to acknowledge, or is otherwise not eligible to be acknowledged, the acknowledge request is discarded for that tag. Any valid requests however are processed.

Example request, the client acknowledges alarms "PB2Alarm", "PB1Alarm":

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":829,
"alarms":["zone1", "zone2", "zone3"],
"alarmack":["PB2Alarm", "PB1Alarm"]
}
```

In this example there is no immediate effect on the server in this cycle. The effect of the request may not necessarily be seen immediately. Response:

```
{
"id": "HMIServer demo server",
"stat": "ok",
"msgid": 829,
"timestamp": 1238157917.0,
"alarms": {"PB2Alarm": {"count": 1, "state": "alarm", "timeok": 0.0,
                        "time": 1238138641.5069301},
           "PB3Alarm": {"count": 2, "state": "ackalarm", "timeok": 0.0,
                        "time": 1238138614.233047},
           "PB1Alarm": {"count": 2, "state": "ok",
                        "timeok":1238139638.2538631, "time":
                        1238138638.3152909}}
}
```

Here the client asks for the alarm list again:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":830,
"alarms":["zone1", "zone2", "zone3"],
"alarmack":[]
}
```

On the next cycle, alarm "PB1Alarm" has disappeared, and "PB2Alarm" has changed state from "alarm" to "ackalarm". Response:

```
{
"id": "HMIServer demo server",
"msgid": 830,
"stat": "ok",
"timestamp": 1238157919.0,
"alarms": {"PB2Alarm": {"count": 1, "state": "ackalarm",
                        "timeok": 0.0,"time": 1238138641.5069301},
           "PB3Alarm": {"count": 2, "state": "ackalarm",
                        "timeok": 0.0,"time": 1238138614.233047}}
}
```

9.3 Events

Events are conditions which the operator is notified of, but which is not require to be acknowledged. They are equivalent to entries in a log file. Like alarms, events represent the current state of a condition which is being monitored.

Events differ from alarms in that they represent a historical record of the state of the condition at the time the event was generated, and do not subsequently change. If the condition being monitored changes, this

generates a new event, rather than changing the state of a previously generated event.

Events also differ from alarms in that the operator cannot not acknowledge them, or do anything which directly affects the state of the event.

An event tag is used to identify a *type* of event. It does not however identify when that event occurred. An event tag may be used to associate an event with a text string which describes the event to the operator. The descriptive text string itself however is not part of the protocol.

For example, the event tag "pump1started" may identify an event described as "Pump #1 has started".

When an HMI client requests events it must include the following information:

- **serial** - This the event serial number of the *last* event which was received by the client. The server is being asked for any events which are newer than this serial number.
- **max** - This is the maximum number of events which the client wishes to receive. If the number of available events is greater than this maximum, only the newest of the available events (up to the maximum) will be sent.
- **zones** - This is a list of event zones for which events are being requested. Only events for the requested zones will be sent.

9.3.1 Event Data

When the server returns a response which contains an event, the response will include the following information:

- **event** - This is the event tag name used to identify the particular type of event.
- **serial** - This is the event serial number used to identify the particular occurrence of that event.
- **timestamp** - This is the time at which the event occurred.
- **value** - This is a user defined data value which may be included with the event.

Examples

Request:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":829,
"events" : {"serial" : 12345, "max" : 50, "zones" : ["zone1", "zone2"]}
}
```

Response:

```
{
"id": "HMIServer demo server",
"msgid": 830,
"stat": "ok",
"timestamp": 1238157919.0,
"events" : [{"timestamp": 1237953973.367131,
           "serial": 1237953949,
           "event": "PumpRunning",
           "value": 1}]
}
```

9.4 Alarm History

Alarm history is like events, but is a record of alarms which were active and have become inactive. Alarm history uses the same zone definitions as used by the corresponding alarms. An alarm becomes part of the alarm history when it changes state from active to inactive.

Alarm history is requested in a manner similar to that used for events. The returned data however is slightly different.

9.4.1 Alarm History Data

When the server returns a response which contains an event, the response will include the following information:

- **serial** - This is the alarm history serial number used to identify the particular occurrence of this alarm history message.
- **alarm** - This is the alarm tag name. This is used to identify the particular alarm.
- **state** - This is the current state of the alarm. This is normally "inactive".
- **count** - This is the number of times the alarm went in and out of the alarm state while it was active in this cycle.
- **time** - This is the time at which the alarm became active.
- **timeok** - This is the time at which the alarm condition last become false before becoming inactive.
- **ackclient** - This is the client id of the client which acknowledged the alarm.

Examples

Request:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":829,
"alarmhistory" : {"serial" : 12345, "max" : 50, "zones" : ["zone1", "zone2"]}
}
```

Response:

```
{
"id": "HMIServer demo server",
"msgid": 830,
"stat": "ok",
"timestamp": 1238157919.0,
"alarmhistory" : [{"count": 1,
   "alarm": "pumptrip",
   "ackclient": "HMI 9876 from Water Pressure INC.",
   "timeok": 1237953984.882,
   "state": "inactive",
   "time": 1237953982.154,
   "serial": 1237953948}]
}
```

9.5 Event and Alarm History Serial Numbers

Each event or alarm history message has a unique serial number applied to it by the server. This serial number is used to track the event or alarm history message throughout its life. This serial number is an integer which must increase for each subsequent event or alarm history message. Serial numbers are guaranteed to be unique and each subsequent serial number is guaranteed to be larger than any previous one. However, they are not guaranteed to be consecutive. That is, there is no guaranty that some numbers in the sequence will not be skipped.

Event serial numbers are independent of alarm history serial numbers. There is no guaranty that any particular event serial number will not also be used as an alarm history serial number.

Serial numbers are always generated by the server. Serial numbers are always positive integers, but the starting point, range, and amount incremented each time is implementation dependent.

9.5.1 Using Serial Numbers in Requests

An HMI client requests new events or alarm history messages by requesting those which are *newer* than a specified serial number. The server will send any events or alarms history messages which are newer than that serial number (up to a maximum specified by the tag "max"). If the server does not have any events or alarm history messages newer than that serial number, it will not send any event or alarm history messages in its response. A serial number is *newer* than another when it has a larger value.

The sequence is as follows:

1. On start-up a client does not know what the latest serial number in the server is. It therefore should make its first request with a very low number, such as 0.
2. The server will respond with the most recent messages, up to the limit specified in "max". The server may respond with fewer messages than specified by "max" if there are fewer messages available, or if it is desired to limit the size of server responses to some lower quantity.
3. The client will receive the messages, and display them to the operator in some suitable fashion.
4. The client will record the largest serial number in the response for use in the next request.
5. In the client's next request, the recorded serial number will be used to request any messages newer than those sent in the previous response. If there are no new messages, then none will be sent in the next response.
6. When new messages arrive, the new largest serial number is recorded for use in the next transaction.
7. The request/response process repeats as described above.

A client will typically display events and alarms to an operator in chronological order in a table or list using the information provided by the message.

Examples

The client begins the communications cycle by sending the first request. Request:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":0,
"events":{"serial":0, "max":50, "zones":["zone1", "zone2", "zone3"]},
"alarmack":[]
}
```

The server sends a response containing the requested data. This includes the current contents of the event buffers. Response:

```
{
"id": "HMIServer demo server",
"msgid": 0,
"stat": "ok",
"timestamp": 1238156675.0,
"events": [{"timestamp": 1238138583.9420061, "serial": 1238138527,
            "event": "PumpRunning", "value": 1},
           {"timestamp": 1238138583.9420061, "serial": 1238138528,
            "event": "PumpStopped", "value": 0},
           {"timestamp": 1238138585.453279, "serial": 1238138529,
            "event":"Tank1Empty", "value": 0},
           {"timestamp": 1238138585.453279, "serial": 1238138530,
            "event": "Tank2Full", "value": 0},
           {"timestamp": 1238138591.5239921, "serial": 1238138531,
```

```
            "event":"PumpRunning", "value": 0},
           {"timestamp": 1238138591.5239921, "serial": 1238138532,
            "event": "PumpStopped", "value": 1}],
}
```

The client now includes the highest event serial numbers in its next request. Request:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":1,
"events":{"serial":1238138532, "max":50, "zones":["zone1", "zone2",
          "zone3"]},
}
```

The events are omitted in the response as there are no events or alarm history records newer than the requested serial numbers. Response:

```
{
"id": "HMIServer demo server",
"msgid": 1,
"stat": "ok",
"timestamp": 1238156676.0,
}
```

9.6 Alarm and Event Zones

A "zone" is a group of alarm or event tags which are related to one another. A client requests alarm, alarm history, or events by requesting zone tags. Configuration and grouping of alarm and event tags into zones is the responsibility of the server.

A zone does not refer to a specific server data table address, memory location, or database record. Rather, a zone is a group of events or alarms. When a client requests information about a zone, it is requesting information about the events or alarms within the zone. Zones allow a client to request only those events or alarms which are of interest to it.

9.6.1 Zone Names

The tag names used for zones must follow the rules used for other tags names. However, alarm and event tags are independent of each other. That is, it is permitted to use the same tag name for an alarm as for an event with any conflict between the two. The server will recognize the zone by the context it is used in. This means that for example alarm zone "pumphouse", event zone "pumphouse", and address tag "pumphouse" are separate and do not conflict with each other.

Event and alarm zone tags are independent of each other. The tag name used for an event zone may also be used for an alarm zone without conflicting with one another. Zone tags are also independent of address tags. A zone tag may be the same as an address tag without causing conflicts.

9.6.2 Assigning Alarms and Events to Zones

The same alarm or event tag name may be used in multiple zones. This means that for example the alarm "tankoverflow" may be used in both zone "pumphouse" and zone "pipeline". A request for either zone will return the correct result for "tankoverflow".

An individual event or alarm may belong to multiple zones. However, each event or alarm must be belong to at least one zone or it will not be addressable by a client.

A client may request multiple zones. If the same alarm or event tag is used in more than one requested zone, the server is to remove any duplicates before returning a response. This means for example that if alarm zones "pumphouse" and "pipeline" are requested, and they both contain the alarm tag

"tankoverflow", then only one copy of the "tankoverflow" alarm is to be returned in the response.

9.7 Dealing with Multiple Clients

Event and alarming is compatible with multiple clients, in that requesting messages from the server does not affect any data on the server.

Any client may send an acknowledge request, and the results of a successful request are available equally to all clients. If the client is not permitted to acknowledge the alarm, or the alarm has already been acknowledged when the alarm acknowledge request is received, the request is discarded.

9.8 Alarm Priorities

In very large systems the data server may impose a limit on the number of alarms it will distribute at one time. Any such limit is a server configuration limit, and is implementation dependent. This limit is therefore not described as part of this protocol.

However, in some situations, the number of alarms which may be present may exceed this limit. This may result in situations where the operator will not be able to view all of the alarms which may be active at one time. In this case it is necessary to impose a priority on the order in which alarms are sent. This priority is:

- **1** - Alarms which are in the "alarm" state are to have the highest priority.
- **2** - Alarms which are in the "ackalarm" state are to have the next highest priority.
- **3** - Alarms which are in the "ok" state are to have the lowest priority.

The reason for this priority is to ensure that the alarms which are most likely to require operator action are presented to the operator first.

An example of the effect this would have is as follows. Suppose the server imposed a limit of a maximum of no more than 50 alarms may be sent at once. We shall also suppose that due to an upset in the control system we have 80 spurious and 10 genuine alarms in the "alarm" state. We would observe the following:

1. The first 50 alarms in the "alarm" state are displayed.
2. The spurious alarms return to the "ok" state as the control system restores the correct input conditions.
3. The 10 genuine alarms still in the "alarm" state receive priority over the spurious alarms in the "ok" state, and become visible on the display. Some of the spurious alarms may inherently be automatically removed from the display to accommodate them.
4. The operator acknowledges the alarms. The 10 genuine alarms change to the "ackalarm" state, and the 40 spurious "ok" alarms become inactive.
5. The 10 genuine alarms still in the "ackalarm" state receive priority over the remaining spurious alarms in the "ok" state, and are still visible on the display.
6. The operator acknowledges the alarms. The 10 genuine alarms remain in the "ackalarm" state, and the final 40 spurious "ok" alarms become inactive.
7. The 10 genuine alarms still in the "ackalarm" are still visible on the display.

The effect is that:

- alarms which return to the "ok" state by themselves are automatically moved to the bottom of the queue to make room for alarms which are not in the "ok" state.
- Alarms which are in the "alarm" state or which change from the "ok" state to the "alarm" state will automatically move to the head of the queue and receive priority for display.
- Alarms which are in the "alarm" state and therefore have not been acknowledged by the operator (and therefore may not have been seen yet) receive priority over alarms in the "ackalarm" state, which presumably have been seen by the operator.
- Alarms which are in the "ackalarm" state, and therefore which still represent a condition which has a fault will receive priority over alarms which are in the "ok" state, which represent a condition where the fault is no longer present, and which are presumably less urgent.

10. Status, Errors, and Access Permissions

10.1 Status

When the server sends a response, it includes a "status" code which indicates the overall state of the message. If there were no errors, a status code of "ok" is returned. Any other code indicates an error which must be handled by the client. If more than one status code is applicable, the first applicable status code from the list below is to be used.

#	***Status Code***	***Description***
1	"unauthorized"	The client is not authorized to communicate with the server.
2	"protocolerror"	An error was encountered in the protocol and it could not be accepted. The entire message was discarded by the server.
3	"commanderror"	A request command field provided incorrect or invalid data. Check each possible error response for any errors. Multiple errors are possible. If a request is made using a command which is valid but not implemented by the server, the response should return this status code.
4	"servererror"	An unspecified error has occurred in the server which prevents the request from being completed. The error is not specified in any of the return fields. The causes of this are implementation dependent.
5	"noclistempty"	The client attempted to read using NOC without an NOC list being present in the server.
6	"ok"	No errors.

Examples

The client is attempting to write to a tag which does not exist. Request:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":1422,
"read":["PB1"]
}
```

The server responds with the data and a status code that indicated that no error was encountered:

```
{
"id": "HMIServer demo server"
"msgid": 1422,
"timestamp": 1238158810.0,
"stat": "ok",
"read": {"PB1": 1}
}
```

10.2 Errors

Error codes are returned from a server as part of a response message. Error codes are never sent from the client to the server. See the command reference to see which responses may return an error code.

If the server returns an error code, it must also return an appropriate status in the status field.

Error codes are text strings which identify a protocol error. This error may have occurred due to a transmission problem, a system configuration error, a programming error, or invalid user input.

Only one error code should be returned for each tag which has an error. If more than one error is applicable, then the code returned should be in the first applicable code in the order listed below.

Error codes are returned in response fields which are associated with the command in which the error occurred. The fields used for reporting errors are detailed in the command reference.

#	***Error Code***	***Description***
1	accessdenied	The client does not have authorization to access this tag.
2	tagnotfound	The address tag is not recognized by the server.
3	writeprotected	An attempt was made to write to an address which is write protected or otherwise not writable.
4	badtype	The data value is of an incompatible type and cannot be converted to the desired type.
5	outofrange	The data value is out of range. Range limits may be imposed by the server based on data type or through configuration limits imposed on a per-tag basis.
6	addresserror	An error occurred in attempting to map the tag to the internal server address representation.
7	servererror	An unspecified error has occurred in the server which prevents the request from being completed.

Examples

The client is attempting to write to a tag which does not exist. Request:

```
{
"id":"HMI 9876 from Water Pressure INC.",
"msgid":1422,
"write":{"PBBad":1},
}
```

The server responds with a "commanderror" in the "stat" field, and a "tagnotfound" in the "writeerr" field.

```
{
"id": "HMIServer demo server"
"msgid": 1422,
"timestamp": 1238158810.0,
"stat": "commanderror",
"writeerr": {"PBBad": "tagnotfound"}
}
```

10.2 Controlling Client Read and Write Access

It is sometimes desirable to control which tags may be read or written to on a tag by tag basis, possibly discriminating between clients. Access control is a server configuration function, and not part of the protocol definition. However, the protocol does define the following responses to failed access attempts:

- If the client does not have any authorization to communicate with the server, the server should return a status of "unauthorized". All unauthorized reads, writes, and other operations are to be ignored by the server.
- If the client is authorized to communicate with the server, but it attempts to read an address tag which exists but for which it has no access permissions, the server should return an error for that tag of "accessdenied" and a status of "commanderror".

- If the client attempts to write to an address tag for which it has read but not write authorization, the server should return an error for that tag of "writeprotected" and a status of "commanderror".
- If the client attempts to read events, alarms, or alarm history without authorization, the server should return an error for that zone of "accessdenied" and a status of "commanderror". Access is defined on a zone basis.
- If the client attempts to acknowledge an alarm for which it does not have acknowledge authorization, the acknowledge request for that alarm tag should simply be discarded without generating an error.

In all cases, unauthorized operations are not to be performed.

Access control granularity is defined as follows:

- **Tag Read and Write** - For address tag read and write operations using "read", "write", and "readnoc", read and write access is defined for each individual tag.
- **Alarms, Events, and Alarm History** - For alarms, events, and alarm history, read access is defined on a zone basis. That is, all alarms, events, or alarm history data which are in the same zone have the same access permissions. Since alarms and alarm history share the same zone definitions, the same zone access applies to both.
- **Alarm Acknowledge** - For alarm acknowledge operations, access is defined on an alarm tag basis. That is, each individual alarm tag may have a separate write access permission level applied to it by the server.

Since individual alarms and events may simultaneously belong to several different zones it is possible for an alarm or event to not be accessible via one zone, but still be accessible via another zone. This allows the same an alarm or event to be part of both a zone with restricted access while still being accessible to non-restricted users via another zone.

The ability of each user to to acknowledge each individual alarm can be restricted by setting the access levels for each individual alarm tag. Since alarm acknowledge does not use zones, this restriction is independent of whatever read access levels have been applied to the alarms.

Examples

The client attempts to communicate without authorization. Request:

```
{
"id" : "Unauthorized client #1",
"msgid" : 12345,
"read" : ["PB1", "PB2", "LS1", "SS5", "TankLevel"],
"write" : {"SOL1" : 0, "SOL2" : 1, "Pump1Speed" : 2250}
}
```

Response:

```
{
"id"    : "Scada 10934 from Water Pressure INC.",
"msgid"  : 12347,
"timestamp": 129873294,
"stat" : "unauthorized"
}
```

The client attempts to read a tag ("TankLevel") without authorization. Request:

```
{
"id" : "Unauthorized client #1",
"msgid" : 12345,
"read" : ["PB1", "TankLevel"],
}
```

Response:

```
{
"id"    : "Scada 10934 from Water Pressure INC.",
"msgid"  : 12347,
"timestamp": 129873294,
"stat" : "commanderror",
"read" : {"PB1" : 1},
"readerr" : {"TankLevel" : "accessdenied"}
}
```

The client attempts to write to a tag ("SOL1") without authorization. Request:

```
{
"id" : "Unauthorized client #1",
"msgid" : 12345,
"read" : ["PB1", "TankLevel"],
"write" : {"SOL1" : 0, "SOL2" : 1}
}
```

Response:

```
{
"id"    : "Scada 10934 from Water Pressure INC.",
"msgid"  : 12347,
"timestamp": 129873294,
"stat" : "commanderror",
"read" : {"PB1" : 1, "TankLevel" : 236},
"writeerr" : {"SOL1" : "writeprotected"}
}
```

The client attempts to write to read "events" without authorization. Request:

```
{
"id" : "Unauthorized client #1",
"msgid" : 12345,
"events" : {"serial" : 12345, "max" : 50, "zones" : ["all"]}
}
```

Response:

```
{
"id"    : "Scada 10934 from Water Pressure INC.",
"msgid"  : 12347,
"timestamp": 129873294,
"stat" : "commanderror"
"eventerr" : {"all" : "accessdenied"}
}
```

---

11. Command Reference

The following commands are defined:

11.1 id

The client and server both send their own id strings and do not echo the other party's id.

Valid for	Request, response
Command data	String
Description	An arbitrary string. This may appear in a request or response and is normally used to identify the client or server. This may be an empty string.
Errors	None. Values with invalid data types are discarded by the recipient. Strings exceeding the maximum allowed string length are truncated by the recipient.
Example Request	"id" : "HMI 9876 from Water Pressure INC."
Example Response	"id" : "Scada 10934 from Water Pressure INC."

11.2 msgid

Valid for	Request, response.
Command data	Positive unsigned 16 bit integer (0 - 65535)
Description	This is used to identify a message. The msgid is generated by the client, sent in a request to the server, and echoed back by the server in its response. This should normally be incremented by the client with each request, returning to zero when it reaches its maximum value.
Errors	None. Values with invalid data types or out of range values are discarded by the recipient. If the server receives an invalid msgid from a client it should respond with a msgid of 0.
Example Request	"msgid" : 12345
Example Response	"msgid" : 12345

11.3 stat

The "stat" response is read by the client to determine if there were any errors in its request. Only one status code may be sent in any response.

Valid for	Response
Command data	String
Description	Returns the overall status of the server response to the client request.
Errors	N/A
Status Codes	See the section on "Status".
Example Request	N/A
Example Response	"stat" : "ok"

11.4 timestamp

Valid for	response
Command data	Time
Description	A value of type "Time", representing the current time as UTC (GMT).
Errors	None. Values with invalid data types or out of range values are discarded by the recipient.

Example Request	N/A
Example Response	"timestamp": 129873294

11.5 read

This is the command normally used to read data from the server.

Valid for	Request, response
Command data	Array (request), Object (response)
Description	In a request, this is an array (list) of address tags for which the client wishes to receive the current data values. In a response, this is an object (dictionary) of key/value pairs containing the tags and their corresponding data values.
Data tags	All data tags will be address tags and will correspond to the address tags sent in the request.
Data values	Each data tag returned will include a data value of the type corresponding to the type defined by the server for that tag.
Errors	If an error occurs, the affected tag will not be returned in the response, but will be returned with an error code in the "readerr" command field.
Example Request	"read" : ["PB1", "PB2", "LS1", "SS5", "TankLevel"]
Example Response	"read" : {"PB1" : 1, "PB2" : 0, "LS1" : 0, "SS5" : 1, "TankLevel" : 50.67}

11.6 readnoc

Valid for	Request, response
Command data	Array (request), Object (response)
Description	This command is identical to the normal "read" command, but it also causes the server to "remember" the list of tags. Subsequent requests using the "readnoc" command with an empty list will return only the changes since the last read.
Data tags	All Data tags will be address tags and will correspond to the address tags sent in the request.
Data values	Each data tag returned will include a data value of the type corresponding to the type defined by the server for that tag.
Errors	If an error occurs, the affected tag will not be returned in the response, but will be returned with an error code in the "readerr" command field.
Example Request	• "readnoc" : ["PB1", "PB2", "LS1", "SS5", "TankLevel"] • "readnoc" : []
Example Response	"readnoc" : {"PB1" : 1, "PB2" : 0, "LS1" : 0, "SS5" : 1, "TankLevel" : 50.67}

11.7 readerr

Valid for	Response
Command data	Object

Description	This is used to report errors from the "read" command. This is an object (dictionary) of key/value pairs containing the tags used in the previous "read" request and their corresponding error codes. Only tags for which an error was encountered are returned. If there were no errors, the response is empty or not present.
Data tags	All data tags will be address tags and will correspond to the address tags sent in the request.
Data values	Each data tag returned will include a data value consisting of a valid error code.
Errors	N/A.
Example Request	N/A
Example Response	"readerr" : {"PB1" : "tagnotfound", "TankLevel" : "servererror"}

11.8 write

This is the command normally used to write data to the server. It only needs to be sent when there is new data to write.

Valid for	Request
Command data	Object
Description	This is a object (dictionary) containing key/value pairs. The request contains values which the client wishes to write to the server.
Data tags	All Data tags will be address tags.
Data values	Each data tag sent must include a data value compatible with the type defined by the server for that tag. The server may convert or rescale the data value (this is implementation dependent).
Errors	If an error occurs, the affected tag will be returned with an error code in the "writeerr" command field. If an error has occurred, the write operation may not have completed.
Example Request	"write" : {"SOL1" : 0, "SOL2" : 1, "Pump1Speed" : 2250}
Example Response	N/A

11.9 writeerr

Valid for	Response
Command data	Object
Description	This is used to report errors from the "write" command. This is an object (dictionary) of key/value pairs containing the tags used in the previous "write" request and their corresponding error codes. Only tags for which an error was encountered are returned. If there were no errors, the response is empty or not present.
Data tags	All data tags will be address tags and will correspond to the address tags sent in the request.
Data values	Each data tag returned will include a data value consisting of a valid error code.
Errors	N/A.
Example Request	N/A
Example Response	"writeerr" : {"SOL1" : "tagnotfound", "Pump1Speed" : "servererror"}

11.10 readable

Valid for	Request, response
Command data	Object
Description	This is an object (dictionary) containing key/value pairs. The keys represent tags, and the values represent requested types (request) or status (response). The client provides a list of keys and associated types it would like the server to examine. If there are any errors, the server returns the key which represents the error along with an error code. If there were no errors, the response is empty or not present.
Data tags	All Data tags will be address tags. Data tags exist both in the request and in the response.
Errors	All errors are returned within the response.
Example Request	"readable" : {"PB1" : "boolean", "LS1" : "boolean", "SS5" : "boolean", "TankLevel" : "integer"}
Example Response	"readable" : {}
Example Response	"readable" : {"PB1" : "tagnotfound", "TankLevel" : "typeerror"}

11.11 writable

Valid for	Request, response
Command data	Object
Description	This is an object (dictionary) containing key/value pairs. The keys represent tags, and the values represent requested types (request) or status (response). The client provides a list of keys and associated types it would like the server to examine. If there are any errors, the server returns the key which represents the error along with an error code. If there were no errors, the response is empty or not present.
Data tags	All Data tags will be address tags. Data tags exist both in the request and in the response.
Errors	All errors are returned within the response.
Example Request	"writable" : {"SOL1" : "boolean", "SOL2" : "boolean", "Pump1Speed" : "integer"}
Example Response	"writable" : {}
Example Response	{"SOL1" : "tagnotfound", "SOL2" : "readonly", "Pump1Speed" : "typeerror"}

11.12 events

Valid for	Request, response
Command data	Object
Description	This is an object (dictionary) containing key/value pairs. Events are messages concerning unique occurrences which the operator is notified about, but which do not require acknowledgement. Each event has a serial number which uniquely identifies it.
Data tags	See below
Errors	N/A
Example Request	"events" : {"serial" : 1223601864, "max" : 50, "zones" : ["zone1"]}

Example Response	"events" : [{"timestamp": 1237953973.367131, "serial": 1237953949, "event": "PumpRunning", "value": 1}]

11.12.1 Event Request Tags

"serial"	This is a unique integer serial number applied by the server. Each newer event must have a higher serial number than each preceding event. When a client sends a serial number in a request, it is requesting only events with higher serial numbers than that specified. The client does not need to know what serial numbers exist on the server, as it is just requesting any greater than the number sent.
"max"	This is an integer sent by a client in a request to indicate the maximum number of responses desired. The server is to respond with no more than this number of events. If the number of available events is greater than this limit, the server will send only the newest events.
"zones"	This is a list of zone tags for which the client wishes to receive events.

11.12.2 Event Response Tags

"serial"	This is a unique integer serial number applied by the server.
"timestamp"	This is a time value applied by the server indicating when the event occurred.
"event"	This is a string tag which is used to identify an event message text. The text of event messages is not sent as part of the protocol, but must be supplied by the client.
"value"	This is a user defined value which may be used to carry additional information relating to the event. This may contain simple types, arrays, or objects. It is up to the client to extract and interpret the data.

11.13 alarms

Valid for	Request, response
Command data	Array (request), Object (response)
Description	Alarms requests specify a list of alarm zones for which the client wishes to receive alarms. The response is a list (array) of alarm objects. Multiple alarm messages may be sent in a single response.
Data tags	See below.
Errors	N/A
Example Request	"alarms" : ["zone1", "zone2", "zone3"]
Example Response	"alarms" : ["PumpAlarm": {"count": 1, "state": "alarm", "timeok": 0.0, "time": 1237956071.725}]

11.13.1 Alarm Response Tags

"state"	This is a string indicating the current alarm state (see below).
"timestamp"	This is a time value applied by the server indicating when the alarm became active.
"timeok"	This is a time value applied by the server indicating when the alarm last became inactive. This is zero (0 or 0.0) whenever the alarm is active.

"count"	This is an integer indicating the number of times an alarm entered the "alarm" state before becoming inactive.

11.14 alarmack

Valid for	Request
Command data	Array
Description	This is an array (list) containing alarm tags for alarms which are to be acknowledged.
Data tags	N/A.
Errors	N/A
Example Request	"alarmack" : ["pumptrip", "tankoverflow"]
Example Response	N/A

11.15 alarmhistory

Valid for	Request, response
Type	Object
Description	This is an object (dictionary) containing key/value pairs. Alarm history is a series of messages containing information about alarms which have become inactive. Each alarm history message has a serial number which uniquely identifies it.
Data Keys	See below
Errors	N/A
Example Request	"alarmhistory" : {"serial" : 1223601864, "max" : 50, "zones" : ["zone1"]}
Example Response	"alarmhistory" : [{"count": 1, "alarm": "pumptrip", "ackclient": "HMI 9876 from Water Pressure INC.", "timeok": 1237953984.882, "state": "inactive", "time": 1237953982.154, "serial": 1237953948}]

11.15.1 Alarm History Request Data Keys

"serial"	This is a unique integer serial number applied by the server. Each newer alarm history message must have a higher serial number than each preceding alarm history message. When a client sends a serial number in a request, it is requesting only alarm history messages with higher serial numbers than that specified. The client does not need to know what serial numbers exist on the server, as it is just requesting any greater than the number sent.
"max"	This is an integer sent by a client in a request to indicate the maximum number of responses desired. The server is to respond with no more than this number of messages. If the number of available messages is greater than this limit, the server will send only the newest messages.
"zones"	This is a list of zone tags for which the client wishes to receive messages.

11.15.2 Alarm History Response Data Keys

"serial"	This is a unique integer serial number applied by the server.

"time"	This is a time value applied by the server indicating when the alarm became active.
"timeok"	This is a time value applied by the server indicating when the alarm last became inactive.
"alarm"	This is a string tag which is used to identify an alarm message text. The text of alarm messages is not sent as part of the protocol, but must be supplied by the client.
"ackclient"	This is the name of the client which acknowledged the alarm.
"state"	This is the state of the alarm at the time it entered the history record.
"count"	This is an integer indicating the number of times an alarm entered the "alarm" state before becoming inactive.

---

12. Revision History

- Draft 2009-03-24. This draft is for review purposes only. This draft includes the "new" style alarms.
- **Draft 2009-05-07. This draft is for review purposes only. This draft includes:**
  - Restructured document.
  - removed "alarm history"
  - reformatted examples.
  - Lines of <80 characters.
  - Removed tabs.
- **Draft 2009-06-05. This draft is for review purposes only. This draft includes:**
  - Continued restructure of document in line with previous changes.
  - Restored alarm history.
  - Added more examples.
  - Added alarm priorities.
  - Changed some titles.
  - Added new introduction.
  - Spell check.
  - Added authors, copyright, license information.
- **Draft 2009-06-06. This draft is for review purposes only. This draft includes:**
  - Added "readerr" and "writeerr" descriptions and examples to "Basic Read / Write Operations".
  - Fixed list formatting in "3.2 Data Tags" to make it consistent with other lists.
  - Fixed capitalisation in several headings to make them consistent.
- **Draft 2009-12-16. Initial public draft release:**
  - Fixed logo size.
  - Bumped date.