

**ZAP**

**ZIOS Application Protocol**

Document Owner:

Dave Rathnow, Software Architect

Document Version:

0.2

Revision History:

|  |  |  |  |
| --- | --- | --- | --- |
| 0.1 | Dave R | July 10, 2013 | Initial version |
| 0.2 | Andrew J | July 16, 2013 | Renamed protocol Defined an encryption type table.  Defined 0=none, 1=AES-128/CTR mode |
| 0.5 | Dave Rathnow | Nov 22, 2013 | Added BundledReportMessage defintion |
| 0.6 | Dave Rathnow | May 30, 2014 | Added Write IO Point message |
|  |  |  |  |

This document describes the ZIOS Application Protocol (ZAP) used to communicate between the field devices and the server.

# Basics

## Byte Order

Byte order used by ZAP will be Big Endian.

## Data Types

### Primitive Types

|  |  |
| --- | --- |
| Type | Size (bits) |
| Byte | 8 |
| Unsigned Byte | 8 |
| Int16 | 16 |
| Unsigned Int16 | 16 |
| Int32 | 32 |
| Unsigned Int32 | 32 |
| Int64 | 64 |
| Unsigned Int64 | 64 |
| Float | 32 |
| Double | 64 |

### Timestamp

32 bit, unsigned value representing an offset from January 1, 1970.

### String Type

Strings are represented as a structure that includes a two byte length field (unsigned) followed by the string contents. The length field does not include the two byte length field:



Example: The following shows how the string “Hello World” would be encoded.



### BLOB Type

Binary Large Objects (BLOB) types are represented as a structure that includes a four byte length field (unsigned) followed by the bytes that make up the BLOB. The length field does not include the four byte length field:



## Authentication

Before any data can be exchanged between the bridge and the field device, the field device must authenticate itself by passing an Authentication Request Packet. This packet identifies the device and provides additional communication options that indicate how data will be transmitted. In the following descriptions, unless otherwise stated, all values are unsigned.

### Hashing Function

The hashing algorithm used is SHA1. The hash is applied to an array of bytes constructed by appending all values. In the following section, the notation used for hashing is

hash(value1, value2,…)

The order of each value is important as the same hashing must be done by both the server and the device in order to calculate the correct hash value. In addition, for added security, the hash algorithm must be repeated exactly 1024 times. For example, if we have three values (value1, value2, value3) and we want to calculate an SHA1 value, the pseudo code would be:

value\_array = append(value1, value2, value3)  
hash\_value = SHA1(value\_array)  
repeat 1023 time:  
 hash\_value = SHA1(hash\_value)

### Authentication Process

The following diagram shows the flow of authentication challenge/response between the server and the field device. The following is assumed:

* The field device is assigned a unique *username* that identifies it to the server
* The server contains a database keyed by *username*
* The field device has a secret key value that is unique to the device
* The server has the device’s secret key value stored along with the username
* The device contains a unique *expected server name* value that identifies the server it will communicate with.



Figure 1 - Authentication Flow

1. Device opens socket to server
2. Server issues challenge
3. Server accepts connection and generates random 16 byte salt value (*server\_salt*) and transmits this value to the device. The salt value is randomly generated array of bytes. The packet sent is exactly 16 bytes in length. (see Figure 2 - Server Challenge Packet)
4. Device responds to challenge
5. Device generates random 16 byte salt value (*client\_salt*).
6. Device creates *client\_hash* = hash(*username, secret\_key, client\_salt, server\_salt).*
7. Device transmits username, client\_salt, client\_hash, requested encryption type, and requested compression type. (see Figure 3 - Device Challenge Response Packet)
8. Server Authenticates
9. Server creates hash\_value = hash(*username, secret\_key, client\_salt, server\_salt)* (*server\_salt* from step 2.a). Compare this value to the client\_hash transmitted in step 3.c.
10. Generate server\_hash = hash(*server\_name, secret\_key, server\_salt, client\_salt)*.
11. Generate *crypt\_session\_key* using random 16 byte value and *secret\_key*. This crypt\_session\_key will be used as the encryption key for the life of the connection. (see Figure 4 - Server Authentication Packet)
12. Sever check to see whether outgoing requests are pending for this device
13. Server transmits *server\_name, server\_hash,* and *crypt\_session\_key*. (see Figure 4 - Server Authentication Packet)
14. Device Verifies Server
15. Device calculates hash\_value = hash(server\_name, secret\_key, server\_salt, client\_salt) and compares it to the server\_hash received from step 4.e.
16. Device verifies the server\_name matches its expected server\_name.

### Authentication Packet Format

#### Server Challenge Packet

The server challenge packet is issued immediately after accepting an incoming connection request from a device. It consists of 16 bytes and has the following format:



Figure 2 - Server Challenge Packet

#### Challenge Response Packet

The device response packet is issued by the device in response to the server challenge packet.



Figure 3 - Device Challenge Response Packet

|  |  |  |
| --- | --- | --- |
| Field | Size (Bytes) | Description |
| Length | 1 | Length of the packet. Does not include the first (this) byte. |
| Encryption Type | 1 | Indicates the type of encryption to be used for the session. Must be zero (no encryption) |
| Compression Type Requested | 1 | Indicates the type of compression to be used for this session. Must be zero (no compression) |
| Client Salt Length | 1 | Length of the client salt field |
| Client Salt | Variable | Salt value generated by the client |
| Client Hash Length | 1 | Length of the client hash field |
| Client Hash | Variable | Client hash value |
| Username Length | 1 | Length of the username field |
| Username | Variable |  |

Table 1 - Challenge Response Packet Fields

|  |  |
| --- | --- |
| Encryption Type | Description |
| 0 | No encryption. An empty crypt session key will be returned. |
| 1 | AES-128/CTR mode. A 16-byte crypt session key will be returned. |

Table 2 – Encryption Types

#### Server Authentication Response

The server authentication packet is sent by the server in response to a authentication request by a device. It has the following format:



Figure 4 - Server Authentication Packet

The length of this packet is determined by the value of the *connection flags.* See the description below.

|  |  |  |
| --- | --- | --- |
| **Field** | **Size (Bytes)** | **Description** |
| Length | 1 | Length of the packet, not including this byte. |
| Connection Flags | 1 | A set of bits that indicate aspects of the connection request. See  Figure 5 - Connection Flags |
| Server Name Length | 1 | Length of the Server Name field |
| Server Name | Variable | Name of the server that granted the authentication request |
| Server Hash Length | 1 | Length of the Server Hash |
| Server Hash | Variable | The Server Hash field |
| Crypt Session Key Length | 1 | Length of the Crypt Session Key field. If the remote device does not request any encryption, this value will be zero. |
| Crypt Session Key | Variable | Key used to encrypt data for this session. If the remote device does not request any encryption, this field will not be present. |

Table 2 - Authentication Response Packet fields

#### Connection Flags

  
Figure 5 - Connection Flags

|  |  |
| --- | --- |
| **Name** | **Description** |
| AUTH | Indicates whether or not the authentication request was successful (1) or failed (0). If the request failed (0) then this will be the last byte in the packet. No other information will be provided. If the request was successful, then the rest of the packet can be interpreted. |
| OUTDATA | If this bit is set (1) then it indicates that there is outbound data waiting for this device. In this case, the device should delay closing the connection to give the server time to marshal and send outbound data. If the bit is not set (0), then no data is queued. |
| DELAY | If set (1), the device should immediately disconnect and reconnect at a later time. The time until the next connection is determined by the device. |
| Bits 3 – 7 | Reserved for future use. Must be zero |

## Packet Format

A ZAP Packet is the packet format used to send and receive data between the Bridge and the field device. A ZAP packet consists of two parts: A switch header and message. The switch header contains information that indicates how to interpret the header and the containing message. The message portion contains a ZAP Message.



Figure 6 - General Packet Format

The Packet Header contains four bytes:



Figure 7 - Packet Header

|  |  |  |
| --- | --- | --- |
| **Name** | **Size (Bytes)** | **Description** |
| Version Number | 1 | Identifies the packet format type. Allowable values are   |  |  | | --- | --- | | Value | Name | | 1 | Session Header | |
| Message Type | 2 | The type of message contained in the packet. |
| Session ID | 2 | Session ID (see Session Management) |
| Sequence Number | 2 | Message sequence number (see Session Management) |

### Session Management

Sessions can be thought of as virtual channels that are multiplexed over a single communication socket. A session is used to separate messages destined for different operations that take place concurrently over the same connection. Sessions are distinguished by a 2 byte integer value in the range of 1-65535 and are represented in packet headers by a 2 byte field. Session ID zero is reserved.

At present, session IDs are always assigned by the bridge whenever it initiates an operation that requires a command-response interaction with the device. Session ID are unique to a connection, meaning that when a connection is broken, the session ID is lost and could be reused when a new connection is established.

Message tracking can be accomplished by using the *Sequence Number* field of the packet header. Sequence numbers are unit to a session and should be incremented for each message sent. This includes unsolicited message that use session ID zero. It provides a method that the receiver can use to detect duplicate messages.

#### Unsolicited Message (Session ID 0)

A session ID of zero is reserved for *unsolicited message*. An unsolicited message is a message originating from the field device as a result of an action or event that occurs on the device. Its delivery to the server only requires that the server send a confirmation that the message was *received and processed*. For example, when a regularly schedule report is generated on the device, it will be sent in using a session ID of zero. The server will accept the message process it and then send back a response indicating that the message was processed. The format of the response is dependent on the message received by the server.

In the case of an unsolicited message, the *Sequence Number* field is used to correlate a response with a message. When the device sends an unsolicited message to server, it assigns a value to the Sequence Number field. When the server sends back a response, it will set the Sequence Number field of the response packet to the same Sequence Number pass by the device. This way the field device can easily match responses with their associated message.

## Framing

Packets are framed using a simple counted byte framing layer. Frames start with a two byte packet that is a count of the number of bytes in the frame, not including the two byte frame count:



Figure 8 - A Frame

# Message Passing

Passing messages between device and server is done using a command-response pattern: one side sends a message and the other is responsible for sending a response indicating the message was received, processed, and, if necessary, what the results of the process were. This process is relying on the reliability of TCP to ensure messages are delivered to their remote destination. It also means that it is the responsibility of the application to maintain state and timing to ensure operations do not wait indefinitely for an response.

# Message Format

|  |  |
| --- | --- |
| **Message Name** | **Type Number** |
| Server Challenge | 1 |
| Challenge Response | 2 |
| Authentication Response | 3 |
| Bundled Report | 4 |
| Write IO Point Message | 5 |
| Demand Poll | 6 |
| Configure | 7 |
| Acknowledgement | 65535 |
| Heartbeat | 65534 |
| Heartbeat Response | 65533 |
| BackOff | 65532 |
| Device Capabilities | 65531 |

|  |  |  |
| --- | --- | --- |
| Name | Number | Description |
| Scheduled | 0 | Generated as a result of a regularly scheduled event |
| Demand Poll | 1 | Report was generated as a result of a demand poll |

Table 3 - Reason Codes

|  |  |  |
| --- | --- | --- |
| Name | Number | Size (bytes) |
| Unknown | 0 | 0 |
| Discrete | 1 | 1 |
| Byte | 2 | 1 |
| Unsigned Byte | 3 | 1 |
| Short | 4 | 2 |
| Unsigned Short | 5 | 2 |
| Long | 6 | 4 |
| Unsigned Long | 7 | 4 |
| Long Long | 8 | 8 |
| Unsigned Long Long | 9 | 8 |
| Float | 10 | 4 |
| Double | 11 | 8 |
| Blob | 12 | Variable |
| String | 13 | Variable |

Table 4 - Data Types

## Report Message

A Report Message is used to deliver IO point values to the server



|  |  |  |
| --- | --- | --- |
| Name | Size (bytes) | Description |
| Message Sequence Number | 2 | Unique sequence number for this message. |
| Pollset Number | 1 | The pollset number for the IO Points in this message |
| Timestamp | 4 | Time the report was generated. |
| Reason Code | 1 | The reason this report was delivered. See Table 3 - Reason Codes |
| Count | 2 | Number of Report Items contained in this message |
| Report Items | Variable | A variable length field containing Report Items (see Report Item) |

Table 5 - Report Message Fields

## Report Item

A report item contains a single value for an IO point



|  |  |  |
| --- | --- | --- |
| Name | Size (bytes) | Description |
| Item Id | 4 | A unique identifier for this IO point value |
| Alarm Status | 1 | The alarm status for the IO Point |
| Data Type | 1 | Data type for the value (see Table 4 - Data Types) |
| Data Value | Variable | The value for the IO point. The size of this field is determined by the Data Type. |

Table 6 - Report Item Fields

## Report Format

The Report object is used to construct and save a report containing IO point values that will be sent to the server. A simple way to illustrate the structure of a report is to use pseudo-xml to show its contents:

<Report>  
 <ReportHeader>  
 </IOPointTemplates>  
 <IOPointTemplate1/>  
 <IOPointTemplate2/>  
 . . .  
 <IOPointTemplateN/>  
 </IOPointTemplates>  
 <ReportHeader>

< IOPointReadingRecords>  
 <IOPointReadingRecord1/>  
 <IOPointReadingRecord2/>  
 . . .  
 <IOPointReadingRecordN/>  
 </ IOPointReadingRecords>  
</Report>

### ReportHeader

The ReportHeader contains information about the content of the report. It specifies how many IO points are represented, the index of each point, and its data type. The following describes the format of the ReportHeader



Figure - Report Header

The header starts with a single byte version number that indicates the version of the Report. This version number defines how the rest of the Report will be encoded. Currently, the only version number is version 1.

The EventId is used by the server to match the report with a request that generated the report. This field can be zero in cases where a request did not cause the report to be generated, for example in the case of a scheduled report or alarm trigger.

The next field is a 2 byte integer that contains the total number of IO Point Templates contained in the report. An IOPointTemplate is a structure that describes the IO points contained in each of the IOPointReadingRecord of the report (see below).

The next field is the Reading Count that defines the number of Reading Records in the Report.

The last field is the Reason Code that defines the reason the report was generated. See Table 3 - Reason Codes for valid values.

What follows is a collection of IOPointTemplate objects. Each IOPointTemplate describes an IO Point that is part of this report. The IOPointTemplate contains the index number of the IO Point and the data type. The order of these objects is important as it defines the order that readings appear in IOPointReadingRecords.

An IOPointReadingRecord begins with a timestamp that indicates the time the IO Point value was recorded. Following this is a collection IOPointReading objects, containing a reading Flags and a Reading Value for a single IO point. The order of these IO Point Readings must correspond to the order of the IO point indexes specified in the report header by the IOPointTemplate objects.



The Flags field contains additional information about the reading value contained in each of the



The top bit, bit 7, is a null value indicator bit. If this bit is set to 1, it indicates that the value field is null, that is, there are no bytes in the value field. If the value is 0, the value field contains a value as specified in the associated IO Point Template object. Bits 0 to 3 contain the alarm status of the IO point. If bit 7 is not set (zero), bit 6 indicates whether the value field contains a value or is null. This situation will happen if a value could not be obtained and the device wants to indicate this by passing a null value with an alarm status.

### Bundled Report Message

A bundled report message contains a number of Reports with a header that maps the position of each report.

Report Header

The report header contains information about the reports contained within the message:



This is followed by a list of report IDs (4 bytes each). The number of report IDs is equal to the “Report Count” field in the header. Following that will be the reports. An entire Bundled Report Message would look like this:



### AckMessage

Fomat:



The Flags field contains additional information that indicates if additional information is provided in the AckMessage.



|  |  |  |
| --- | --- | --- |
| Bit Field | Name | Description |
| 0 | Additional Details Flag | Bit zero is the additional details Flag. If this flag is set (equals 1) then there are more bytes in the ACK message. These additional details will begin with a two byte length field, followed by a single byte indicating the type of the data:    The length field is the length of the entire additional details field, not including the length. |
| 1 | Protocol Error | Bit 1 indicates a protocol error. If this bit is set, an error was detected while attempting to decode the message. If this bit is set, the additional details flag should also be set and contain a value to indicate the cause of the error. Protocol error details |

### ACK Message Details Types

Bundled Report ACK Details (Details Type = 1)

This field contains the status return when a Bundle Report Message is processed by the server. It contains of ReportId/Status pairs that indicate the status for each report processed. The first byte of the field is a two byte count field that indicates the number of ReportId/Status pairs in the field.



Protocol Error (Details Type = 2)

The details contain a single 4 byte status value indicating an error in the protocol.



Figure 9 - Protocol Error ACK Details

The value of the size field will always be 4.

|  |  |
| --- | --- |
| Error Code | Description |
| 1 | Invalid Session Id. The session ID specified in the message was invalid. Currently, the only invalid session ID is zero (0). |

### Write IO Point Message

A Write IO Point message is used to set the value of one or more IO Points on a device. The message consists of a count field followed by a number of Write Value objects that define the IO Point id, it’s data type, and the value to be written.



Figure 10 - Write Value

|  |  |
| --- | --- |
| Field | Description |
| IOID | 32 bit number that uniquely identifies and IO point on the device |
| Data Type | 1 byte data type that defines the type of data in the value field. Refer to Table 4 - Data Types |
| Value | 0 or more bytes depending on the data type |

A Write IO Point message is a collection of WriteValue objects:



Figure 11 - Write IO Point Value Message

|  |  |
| --- | --- |
| Field | Description |
| Size | Total size of the Write IO Point Message payload. This size field includes the 1 byte count field and all the bytes making up the Write IO Point Value objects. It does not count the 2 bytes for the count field. |
| Count | The number of Write IO Point Value objects included in this message. Will always be greater than zero |
| Write IO Point Value(s) | Variable length field containing one or more WriteValues. It’s length will vary depending on the number of WriteValues it contains. |

#### Write IO Point ACK Details (Details Type = 4)

The acknowledgement to a Write IO point message contains a global ACK value that indicates that the device was able to decode the entire message and a individual ACK value for each WriteValue specified in the message. The global ACK value tells the server that whether or not the entire message was decoded successfully. Success means that the entire message was decoded and that a status code will be provided for all the WriteValues. Failure means that the entire message could not be decoded. The server should check to see if there are any WriteValue ACKS present because the device may have processed some of the WriteValues passd in the WriteIOPointMessage.



Figure 12 - Write IO Point Message ACK

|  |  |
| --- | --- |
| Field | Description |
| Count | A count of the number of WriteValue ACKs contained in the ACK Payload portion of the message. If this value does not match the number of WriteValues sent with original WriteIOPoint message, it means the device could not decode the entire message |
| ACK Payload. | WriteValue ACKs for each WriteValue the device could decode. |



Figure 13 - WriteValue ACK

|  |  |
| --- | --- |
| Field | Description |
| IOID | 32 bit number that uniquely identifies and IO point on the device |
| Status | 0 – Value written successfully  1 – Failure occurred |

### Demand Poll Message

Use the DemandPollMessage to generate to query a single IO Point or Pollset from the device.



|  |  |
| --- | --- |
| Field | Description |
| EventId | The event ID associated with this demand poll. The resulting Report should return this value so that associated event on the server can be updated. This can be specified as zero. |
| Pollset Number | Pollset number |
| Index | Index of individual IO Point |

You can specify a pollset number or index but not both. If one is specified as a non-zero number, the other must be zero.

#### DemandPollMessage ACK Details (Type=5)

ACK details consist of a single byte value.

|  |  |
| --- | --- |
| Value | Description |
| 0 | Success. This indicates that the demand poll was queued inside the device. The report generated as a result of the poll will be delivered at a later time. |
| 1 | Invalid index |
| 2 | Invalid pollset number |

## Request Time Message

The Request Time message is used to request the time from the server.



|  |  |
| --- | --- |
| Field | Description |
| Device Time | The time on the local device. This time will be passed back in the Server Time Message |

## Server Time Message

A Server Time Message is sent in response to a Request Time Message. It has the following format:



|  |  |
| --- | --- |
| Field | Description |
| Device Time | The Device Time field from the Request Time message. |
| Server Time | The current time on the server |

## Scrub Message

The Scrub Message causes the device to delete all or part of its internal configuration.



|  |  |
| --- | --- |
| Field | Description |
| Version | Version of the message |
| Scrub Options | Bit mask that defines what parts of the device’s configuration should be scrubbed. The mask is a logical or between the values:  0x01 – Scrub IO Points  0x02 – Scrub Reports  0x04 – Scrub Events  0x08 – Scrub Ports |

#### Scrub Message Ack Details (Type=8)

The Scrub Message Ack Details contain information about the status of the command after it was executed by the device:



|  |  |
| --- | --- |
| Field | Description |
| Version | Version of the message |
| Status | Indicates if the command was successful or if it failed:  0 – Success  1 – Failure |
| Failed Options | If the Status field indicates a failure (Status = 1), this field is a bit mask that defines what operation failed. The mask is a logical OR of the values:  0x01 – Scrub IO Points Failed  0x02 – Scrub Reports Failed  0x04 – Scrub Events Failed  0x08 – Scrub Ports Failed |

Configure Message

Configure IO Point

Add IO Point

The Add IO Point Configure message contains information necessary to add an IO Point to the device. With the exception of the Alarm Values, all values must be provided in the message. Alarm Values need only be provided for values defined in the Alarm Mask (see below)

# Web Access

ZIOS provides a simple web interface the main device application. It is provided by a WebServer that uses FastCGI to communicate with the ZIOS device application.

Line delimiters

The only line delimiters supported by these services is the two character sequence carriage-return/ line-feed (<CR><LF>, “\r\n”, 0x0D0A)

## URL Mapping

All URLs passed to the web engine must have the following format:

<pathname>/<scriptname>/<action>[?<arg1>%<arg2>…]

Where argx has the format:

<name>=<value>

The pathname is ignored by the web engine as it is used by the server to map Fast CGI requests to the ZIOS web handler. The <scriptname> identifies the script, or in our case, class that will handle the request and the <action> defines which action will be invoked on the class.

## Web Responses

All Web response will be sent back as JSON. The JSON value will be encoded in a two form argumen”. The first argument will have a

{   
 Status: “<status>”,  
 <additional-content>  
 }

Where “<status>” can be one of “ok” or “error”.

# Appendix I – Device Configuration

Device configuration is stored in a JSON text file. Each key contains either a numeric or string value, according to JSON encoding rules. Some keys have special meaning and are noted below.

### Time specifications

Times for configuration key can be specified as either Absolute Time, i.e. a specific time in the past or future, or as a Delta Time, a time period measure in days, hours, minute, and/or seconds.

#### Absolute Time

An Absolute Time is specified as IOS 8610 time format. The format can contain a date, time, or both date and time. The following are valid date formats:

2015-03-26  
 10:53:00  
 2014-02-19T10:10:00  
  
if only a date is provided, the time is assumed to be “00:00:00”. If only a time is provide, the date is assumed to be 1970-01-01.

#### Delta Time

A Delta Time represents an elapsed period of time. It can be expressed as:

[D] HH:MM:SS

Where: D is optional and represents number of days.  
 HH is the number of hours  
 MM is the number of minutes  
 SS is the number of seconds

Examples:

00:00:30 – 30 seconds

00:01:50 – 1 minute and 50 seconds.  
 10 15:10:40 – 10 days, 15 hours, 10 minutes, 40 seconds.  
 25 36:145:85 – 25 days, 36 hours, 145 minutes, 85 seconds.

### Connection Recycler

The Connection Recycler will close the connection and reopen it at regular intervals.

"Connector" : {  
 "RecycleTimer" : {  
 "Enable" : "true",  
 "Interval" : "00:00:10",  
 "ShutdownWaitInterval" : "00:01:00"  
 }  
}

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Description |
| Enable | Bool | Disables or enables the connection recycler |
| Interval | Delta Time | Maximum time the connector can remain connected to the server before the connection is closed and reopened. The interval is measure from the time the connector connects to the server. If connection is lost for a reason other than the recycle interval expiring, the recycle timer will be reset. |
| ShutdownWaitInterval | Delta Time | The recycler will only disconnect the current connection if there is no activity. The recycler will wait ShutdownWaitInterval for all sessions to close and for the connector to become inactive. If the connector does not become inactive, the connector will be closed and reopened. |

### Web Configuration

Web configuration is done using the “Web” configuration key:

"Web" : {  
 "Enable": "true",  
 "PortNumber" : 5500,  
 ["BackLog" : 50,]  
 ["SessionTimeout" : "01:00:00",]  
 ["LoginTimeout" : "01:00:00",]  
 ["MaxSessions" : 10,]  
 "Users": {  
 "Guest": {  
 "Password": "<encrypted-password>"  
 },  
 "Field": {  
 "Password": "<encrypted-password>"  
 },  
 "Admin": {  
 "Password": "<encrypted-password>"  
 }  
 }  
 }

|  |  |  |
| --- | --- | --- |
| Field Name | Accepted Values | Description |
| Enable | true, false, yes, no | If disabled, the Web Server will not be started. |
| PortNumber | 1 - 65535 | Sets the port number the FIG Web Application server thread will listen on for incoming web requests. |
| BackLog | Unsigned Integer | Optional. Sets the number of incoming web request that can be queued by the FIG Web Application server. If not specified, the default value is 50 |
| SessionTimeout | Delta Time | Optional. Sets the amount of time a Web session can be inactive before it will be deleted. If not specified, the default value is 1 hour. Setting this value to zero will disable session timeout. |
| LoginTimeout | Delta Time | Optional. Sets the amount of time a user is given to login before the session is considered inactive and is deleted. Specifically, it defines the amount of time a user can stay on the login page without login in. If not specified, the default value is 3 minutes. Setting this value to zero will disable login timeout. |
| MaxSessions | Unsigned Integer | Optional. Sets a limit on the number of Web sessions that can be created. If not specified the default value is 10. |
| Users | Static JSON Map | This key defines property for the user types allowed to log into the FIG device. Each user has a password key that is the encrypted value for the user’s password. |

Session Management

A web session is started whenever a user accesses any URL within the ZIOS Web site. Normally, a user should enter at the root web site, which can be accessed by entering http:://<ip-address> into the URL field of your browser. However, entering any other path under this domain will also result in a session being created. To track sessions, a session cookie is created (named “zios.sid”). This cookie is use by the Web subsystem to find an existing session object for each request. If a session cannot be found that matches the current value of zios.sid, a new session will be created and the value of zios.sid will be over written.

Web sessions required system resource so if there is no limit placed on the number of session that can be created, there is a chance that a malicious user could crash the ZIOS application by created an large number of Web sessions. To prevent this from happening, the three configuration keys SessionTimeout, LoginTimeout, and MaxSessions should be configured accordingly.

MaxSessions places a cap on the number of sessions that can be created. Once this limit is reached, an attempt will be made to close any sessions that are inactive. A session is considered inactive depending on whether a session is “unauthenticated” or “authenticated”. Unauthenticated sessions are sessions that have been created but the user has entered username/password. This would be a session where a user has accessed the ZIOS Web UI login page, but has gone no further. Authenticated sessions are those where a user has entered valid username/password.

When MaxSession is reached, and an attempt is made to create a new session, a scan is made of all existing sessions. Any inactive session will be deleted. A session is considered inactive if

1. It is unauthenticated and has been inactive for the period of time specified by LoginTimeout.
2. Is authenticated and has been inactive for the period of time specified by SessionTimeout.

If, after this scan, there are still no session slot available, a scan will be made of all session and the session that has been inactive for the longest period of time, regardless of whether it is authenticated or unauthenticated, will be delete.

### Report Delivery Service Configuration

The Report Deliver Service configuration define configuration parameters for the Report Delivery Service. Example:

"ReportDeliveryService": {  
 "MaxBufferSize": 2048,  
 "RetryStrategy": {  
 "Name": "FixedInterval",  
 "MaxRetries": 3,  
 "Interval": "00:00:30"  
 },  
 "Persistence": {  
 "Directory": "<directory-path>",  
 "MaxFileSize": 64000  
 }  
 }

|  |  |  |
| --- | --- | --- |
| Key Name | Data Type | Description |
| MaxBufferSize | Unsigned Integer | Sets the size, in bytes, of the buffer used to create report messages. Default value is 2048. |
| RetryStrategy | N/A | Define the strategy used to resend report in the event no acknowledgement is received from the server |
| RetryStrategy:Name | Enumeration | Sets the name of the retry strategy. Currently, the only valid value is “FixedInterval”, which means the report will be resent at regular fixed intervals. |
| RetryStrategy:MaxRetries | Unsigned Integer | Sets the maximum number of times a report will be resent before waiting a random amount of time and then attempting to resend the message |
| RetryStrategy:Interval | Delta Time | Specifies the amount of time to wait between failed send attempts. |
| Persistence | N/A |  |
| Persistence:Directory | Directory Path | Sets the location where message persistence files will reside |
| Persistence:MaxFileSize | Unsigned Integer | Set the maximum size of the message persistence file. When this limit is reached, the persistence file will be closed. It will be deleted only if it is in a consistent state. |

## Connector Configuration <TBD>

Connector:WriteTimeoutSeconds

If a write operation does not write all bytes, it will wait this many seconds before aborting the write operation and closing the socket. Default: 20 seconds.

Connector:KeepAliveStrategy

Required: Optional  
Type: Object

This key defines whether or not a keep alive strategy will be used to maintain an active connection to the server. If it is not present, no keep alive strategy will be used. The default strategy will send a heartbeat message to the server at a regular interval. If non-presistent connections are being used, the KeepAliveStrategy should not be used.

Two subkeys can be specified:

|  |  |  |
| --- | --- | --- |
| Key Name | Accepted Values | Comment |
| Enabled | true, false, yes, no | Enables or disables the keep alive timer. |
| HeartbeatSeconds | seconds > 0 | The number of seconds between each heartbeat message. |

Connector: RecycleTimer

Required: Optional  
Type: Object

This key defines how long a persistent connection to the server will be held open before it is closed and immediately reopened. This feature is provided as a means to reset a connection on a regular bassis.

Two sub keys can be specified:

|  |  |  |
| --- | --- | --- |
| Key Name | Accepted Values | Comment |
| Enable | true, false, yes, no | Enables or disables the recycle feature |
| Minutes | Minutes > 0 | The number of minutes to keep the connection open before closing and reopening |
| Seconds | Seconds > 0 | The number of minutes to keep the connection open before closing and reopening. This parameter is provided for testing. If both Minutes and Seconds are present, only Minutes will be used. |

Connector:ControlHandling:TheadPoolSize

Required: Optional  
Type: Integer  
Default Value: 4

Sets the number of thread provided to handle incoming controls. When a message arrives from the server with a session ID that does not map to an existing session, the connector will create a session and hand the session and message off to a thread for handling.

Connector:ControlHandling:ResponseTimeoutSeconds

Required: Optional  
Type: Integer  
Default Value: 20 Seconds

Sets the maximum number of seconds the device will wait for a message to be received from the server during the processing of a control.

Web

Required: Optional  
Type: Object

Defines properties used by the Web listener thread that accepts web requests.

Three sub keys can be specified:

|  |  |  |
| --- | --- | --- |
| Key Name | Accepted Values | Comment |
| Enable | true, false, yes, no | Enables or disables the FastCGI server. |
| PortNumber | 1 - 65535 | The TCP/IP port number the FastCGI process will listen for web request. |
| BackLog | 1 - 65536 | The number of web request that will be queued before they will be dropped. |

## Time Synchronization

The device provides a facility to ensure the system clock is synchronized to the server and can be enabled or disabled through configuration. When enabled, time synchronization will always be done when the device authenticates with the server. Optionally, an event can be configured to synchronize the clock at regular intervals. Configuration of the Time Synchronization feature can be done through the TimeSync key. If the TimeSync key is not present, time synchronization will be disabled.

The following are the configuration parameters for Time Synchronization:

"TimeSync" : {  
 "Enable" : "true",  
 "RoundTripThreshold": "00:00:10",  
 "Interval": "12:00:00",  
 "AdjustTimeThreshold" : "00:00:15",  
 "MaxTimeThreshold" : "00:05:00",  
 "HWClockTimeSetThreshold" : "02:00:00",  
 "DeveloperMode": "foo"  
 }

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| Enable | Boolean | Enabled or disables time synchronization for both login and regular event.  This field is required. |
| RoundTripThreshold | Delta Time | Specifies the maximum amount of time that will be allowed for the server to respond to a Time Request. If the server’s response takes more than this time, the time sync operation will fail and not adjustment to the system time will be done.  This field is required. |
| Interval | Delta Time | Amount of time between each time sync event.  This field is optional. If not present, no time sync event occur, however, time sync will still be done each time the device connects and authenticates with the server. |
| AdjustTimeThreshold | Delta Time | Specifies the minimum amount of difference between the current system time and the server time that will be tolerated before any change to the system time will be done. If the time difference greater than or equal to this value, but less than MaxTimeThreshold, a slow, gradual adjustment to the system time will be done. The system time will be pushed to the HW clock according to the HWClockTimeSetThreshold setting.  This field is required. |
| MaxTimeThreshold | Delta Time | Specifies the maximum amount of difference between the current system time and the server time that will be tolerated before an abrupt change is made to the system time. If the time difference is greater than or equal to this value, the system time will be changed immediately. The system time will be pushed to the HW clock immediately.  This field is required. |
| HWClockTimeSetThreshold | Delta Time | Specifies the amount of time that must elapse before the current system time is pushed to the HW clock whenever a gradual adjustment is made to the system clock. |
| DeveloperMode | Any | When this key is present, time sync operations will occur as usual, but no actual changes will be made to the system or hardware clock. Instead, only log messages will be displayed. |
|  |  |  |

The Time Synchronization feature will maintain both the system clock and the hardware clock. Changes to the system clock will be done according to the settings of AdjustTimeThreshold and MaxTimeThreshold fields.

However, setting the hardware clock is only done at specific times. Each time the HW clock is changed, the time is noted. Any subsequent changes will be done according to the following rules (we will call this *lastTimePush*):

1. If the difference between the current time and the server time is less than AdjustTimeThreshold, the current time will be pushed to the HW clock if the difference between the lastTimePush and the current time is greater than HWClockTimeSetThreshold.
2. If the difference between the current time and the server time is great than or equal to AdjustTimeThreshold, but less than MaxTimeThreshold the current time will be pushed to the HW clock if the difference between the lastTimePush and the current time is greater than HWClockTimeSetThreshold.

If the difference between the current time and the server time is greater than MaxTimeThreshold, the time will be immediately pushed to the HW clock.

# Appendix II – Message Persistence

When a message is queued for transmission, the message is serialized into a persistence file to ensure it is saved in the event that the device is shutdown or crashes before it can be delivered to the server. The message is not deleted from the store until the message is acknowledged as having been received and processed by the server. The persistence file is a sequential file where data is always appended to the end of the file. For this reason, messages are never really truly deleted from the file. Instead, each time a message is written to the file, it is stored as a *Persistence Record* with a unique tracking ID. When an acknowledgement is received from the server, a *Deletion Record* is written that contains the tracking ID. This make the persistence file a *journal* file that contains a history of message sent from the device to, and acknowledged by, the server. A persistence file is said to be *consistent* if for every Persistence Record there is a matching Deletion Record. It is *inconsistent* if there is at least one Persistence Record that does not have a matching DeletionRecord.

The persistence file location is determined by the reading the configuration key “Persistence:Directory”. If the directory does not exist when ZIOS starts, the directory will be created.

The persistence file will not grow forever. The maximum size of the file is determined by the value specified by the configuration key “Persistence:MaxFileSize”. Once the persistence file grows beyond this size, it will be closed and a new file will be opened. The old persistence file will be immediately deleted if it is in a *consistent* state. If the old persistence file is in an *inconsistent* state, ZIOS will wait until it become *consistent* before deleting it.

The format of the persistence file is:

msgXXXXXXXXX.jnl

Where “XXXXXXXXX” is a sequentially increasing hexadecimal number. Users should never change the name of these files nor delete them. Generally, you should only see one file in the persistence directory, however, it is possible to see more than one persistence file in cases where the device has been unable to contact the server for long periods of time and has a large number of messages stored. Once communication with the server has been restored, and all messages age sent and acknowledged, the older files will be deleted.

## Persistence File Format

|  |  |
| --- | --- |
| Record Type Number | Name |
| 1 | Persistence Record |
| 2 | Deletion Record |

### Persistence Record

When a tracking container is inserted into the Tracking Queue, the container is immediately persisted to disk. When a container is persisted to disk, its starting location is stored along with the container so that if it needs to be retrieved later, it can be retrieved using random access operations.



|  |  |
| --- | --- |
| Field Name | Description |
| Flags | Not currently used |
| Length | Length of the serialized message portion of the record. |

### Deletion Record

A Deletion Record indicates that a tracking container was removed from the tracking queue.



# Appendix III

## Configure Event XML

The Configure Event XML follows the standard ZIOS event structure:

<Event name='Configure' qualifier='ZIOS'>  
 <EventId>**[event-id]</**EventId>  
 <Nuid>**[network-unit-id]**</Nuid>  
 <Configure object='**[object-type]**'>  
 <Action type='**[action-type]**'>  
 <CorrelationId>123</CorrelationId>  
 **[Action Parameter Elements]**  
 </Action>  
 . . .  
 </Configure>  
</Event>

The Configure element contains the necessary information to add, delete, or update entities on the device. The XML can contain only one <Configure> event and it must contain an object type attribute.

The <EventId> element is an optional, U64 value. It is optional because the general structure of a ZIOS Event does not require it.

Valid values for the ‘object’ attribute are

|  |
| --- |
| **Object Type** |
| site |
| device |
| ioPoint |
| event |
| port |

A Configure element must contain one or more <Action> elements that define the type of action to invoke for the give object type specified in the <Configure> element. Each <Action> element must contain a ‘type’ attribute that defines the type of action. The valid values for ‘type’ are

|  |
| --- |
| **Action Type** |
| add |
| delete |
| update |

Every <Action> element MUST contain a <CorrelationId> element. This value is used to match actions performed on the device with the actions requested by the server. If any action is missing a <CorrelationId> element, the entire Configure Event to be rejected.

After the <CorrelationId> element, one or more action parameter element can be specified. The types of elements are specific to the object type and action being performed and are documented below.

***Warning: Because the contents of the <Action> element are variable, depending on the object type and action, there is a high degree of flexibility in the creating and modifying objects on the device. However, this flexibility makes it impossible to validate the XML against an XSD. This means that care must be taken to ensure any Action event contains all the information required to configure any object type, for any action. The only validation performed is to validate that a <CorrelationId> element is present.***

### Configure Element XML Example

The following is a sample XML document that configures sites

<Event name=*'Configure'* qualifier=*'ZIOS'*>

<EventId>12345</EventId>

<Nuid>ZED-00001</Nuid>

<Configure object=*'site'*>

<Action type=*'delete'*>

<CorrelationId>123</CorrelationId>

<Id>5</Id>

</Action>

<Action type=*'delete'*>

<CorrelationId>567</CorrelationId>

<Id>21</Id>

</Action>

<Action type=*'add'*>

<CorrelationId>765</CorrelationId>

<Name>Freddy Zipplemier</Name>

</Action>

<Action type=*'update'*>

<CorrelationId>987</CorrelationId>

<Id>456</Id>

<Name>Archie Bunker</Name>

</Action>

</Configure>

</Event>

## Configure Response XML Event

Every Configure Event will have an associated response event that indicates the outcome of the operation on the device.

<Event name=*'ConfigureResponse'* qualifier=*'ZIOS'*>

<EventId>**[even-id]</**EventId>

<Nuid>**[network-unit-id]**</Nuid>  
 <ConfigureResponse object = *'***[object-type]***'*>

<Action type = ’***[action-type]****’*>

<CorrelationId>123</CorrelationId>

<ErrorCode>0</ErrorCode>

[<Id>987</Id>]

[<Message>[error-message]</Message>]

</Action>

<Action type = ’**[action-type]**’>

<CorrelationId>456</CorrelationId>

<ErrorCode>89</ErrorCode>

<Message>[error-message]</Message>

</Action>

. . .

</ConfigureResponse>

</Event>

The Configure Response XML Event mirrors the Configure XML Event. Actions in the Configure Response event are matched with Actions in the Configure Event through their correlation IDs. An Action will always contain a CorrelationID.

The <Id> element will only be provided for successful ‘add’ operation. The value of the <Id> element represents the unique identifier for the new element.

All Actions will contain an <ErrorCode> element to indicate the status of the operation on the server. Zero indicates that the operation was successful. A non-zero ErrorCode indicates an error occurred while validating or processing the error. If a validation error occurs, that is, an error associated with processing or parsing the data prior to executing the operation, a <Message> element will be provided. Errors that occur while executing the operation will result in only an error code being returned with no <Message> element. In these cases, error codes must be statically mapped to their associated text message. (\*\*\*TBD\*\*\* return codes need to be documented).

### Configure Response XML Example

The following is a sample Configure Response XML event that would be published in response to the example given in the Configure Element XML Example section.

<Event name=*'ConfigureResponse'* qualifier=*'ZIOS'*>

<EventId>12345</EventId>

<Nuid>ZED-00001</Nuid>

<ConfigureResponse object=*'site'*>

<Action type=*'delete'*>

<CorrelationId>123</CorrelationId>

<ErrorCode>0</ErrorCode>

</Action>

<Action type=*'delete'*>

<CorrelationId>567</CorrelationId>

<ErrorCode>0</ErrorCode>

</Action>

<Action type=*'add'*>

<CorrelationId>765</CorrelationId>

<ErrorCode>0</ErrorCode>

<Id>6342</Id>

</Action>

<Action type=*'update'*>

<CorrelationId>987</CorrelationId>

<ErrorCode>49</ErrorCode>

<Message>Invalid Value: Name</Message>

</Action>

</ConfigureResponse>

</Event>

## Data Types

Value provided by each action parameter elements must conform to accepted data types. In the following sections, each object type is listed with their associated action and the elements that must be provided. Each element must conform to the data type listed in the following table:

|  |  |
| --- | --- |
| **Name** | **Description** |
| S8 | 8 bits signed |
| S16 | 16 bits signed |
| S32 | 32 bits signed |
| S64 | 64 bits signed |
| U8 | 8 bits unsigned |
| U16 | 16 bits unsigned |
| U32 | 32 bits unsigned |
| U64 | 64 bits unsigned |
| F32 | 32 bit Float |
| String | ASCII String |

## Configure Entity Actions, Elements, and Values.

In the following Sections, unless specified, all value must be present. While not included in their description, all Action elements ***MUST*** contain a CorrelationId.

### Configure Site

#### Add Site

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Name | String |  |

#### Delete Site

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Id | U16 |  |

#### Update Site

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Name | String |  |
| Id | U16 |  |

### Configure IO Point

The configure IO point adds, deletes, or updates IO points on the device.

#### Add IO Point

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| IOPointClass | U8 | Valid value: 1 – RTU  2 – Memory  3 – Configuration  4 – Internal  5 – Network |
| DataType | U8 | Value Values:  1 – Discrete  2 – Char  3 – Unsigned Char  4 – Short  5 – Unsigned Short  6 – Long  7 – Unsigned Long  8 – Float (32 bit)  10 – String |
| PollSetId | U16 |  |
| ExternalDeviceId | U16 |  |
| Tag | String |  |
| SiteId | U16 |  |
| SourceAddress | String |  |
| SensorClassName | String | Optionally can be blank. |
| IsReadOnly | U8 | Boolean value where zero (0) is false, anything else is true. |
| AlarmMask | U32 | Bit mask. If zero, not alarms where specified. If > 0, it represents the logical ORing of the following values:   |  |  | | --- | --- | | **Alarm** | **Mask** | | Low Low Alarm | 0x01 (bit 0) | | Low Alarm | 0x02 (bit 1) | | High Alarm | 0x04 (bit 2) | | High High Alarm | 0x08 (bit 3) |   If any of these values are specified, then the values must provide by associated alarm parameter elements, which follow. The data type of the alarm parameter elements is dependent on the data type of the IO Point being added. |
| AlarmSetHysteresis | U32 | Seconds |
| AlarmClearHysteresis | U32 | Seconds |
| LowLowSet |  | Required if AlarmMask bit 0 is set |
| LowLowHysteresis |  | Required if AlarmMask bit 0 is set |
| LowSet |  | Required if AlarmMask bit 1 is set |
| LowHysteresis |  | Required if AlarmMask bit 1 is set |
| HighHighSet |  | Required if AlarmMask bit 2 is set |
| HighHighHysteresis |  | Required if AlarmMask bit 2 is set |
| HighSet |  | Required if AlarmMask bit 3 is set |
| HighHysteresis |  | Required if AlarmMask bit 3 is set |

#### Update IO Point

The update action must have all the elements descripted in the Add IO Point section, with the addition of an <Id> element that identified the existing IO point to be updated.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Id | U32 |  |

#### Delete IO Point

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Id | U32 |  |

### Configure Device

The usual: Adds, deletes, and updates external devices on the device.

#### Add Device

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Name | String |  |
| External Device Address | U16 | Protocol address |
| PortId | U16 |  |
| ProtocolHandlerName | String | Eg: MODBUS |
| ProtocolArguments | String | JSON encoded map (key/value pairs) |

#### Update Device

The update Device action must contain all the elements from the Add Device Action and an <Id> element to identify the device to be updated.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Id | U16 |  |

#### Delete Device

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Id | U16 |  |

### Configure Event

Adds, deletes, and updates timed events on the device.

#### Add Event

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Name | String |  |
| TimedEventType | U8 | Valid Values:  1 – Poll  2 – Report |
| StartTime | U32 |  |
| Interval | U32 |  |
| Duration | U32 |  |
| PollSetId | U16 | Only required if TimedEventType is 1 or 2. |

#### Update Event

The update Event action must contain all the elements from the Add Event Action and an <Id> element to identify the device to be updated.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Id | U16 |  |

#### Delete Event

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Id | U16 |  |

### Configure Port

#### Add Port

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| PortType | U8 | Valid Values:  1 – RS232  2 – RS485  3 – TCP  4 – Test |
| Name | String |  |
| Parameters | String | Specific to the type of port being added |
| PassThroughPort | U16 |  |
| PassThroughPortTimeout | U32 | sec |
| ModeControl | U8 | Valid values:  1 – None  2 – GPIO  3 – RTS  4 – RTSCTS |
| StartDelay | U32 | Milliseconds |
| EndDelay | U32 | Milliseconds |

#### Update Port

The update Port action must contain all the elements from the Add Port Action and an <Id> element to identify the port to be updated.

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Id | U16 |  |

#### Delete Port

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Notes** |
| Id | U16 |  |