THE ETPORT APPLICATION PROGRAMMING INTERFACE (API)

The ETport and the W2-E4 have an application programming interface (API) that can be used to write custom applications or to re-brand the built-in web interface. Programs or web applications can send HTTP-based requests to the ETport or W2-E4 and it will respond with the desired information or take the desired action. Requests can be made using XML via an HTTP POST, or embedded directly into the URL of an HTTP GET. Requests can also be embedded in a template file, for use with the Web Posting feature of these devices. Responses are returned either in XML, JSON, or plain-text. API requests to modify settings or perform certain operations are password-protected.

This document describes the API itself and describes some aspects of HTTP, XML, and JSON which are used by the API; however, the complete details of HTTP, XML, and JSON are beyond the scope of this document. For full details, please see the following standards:

HTTP: RFC 2616 Hypertext Transfer Protocol – HTTP 1.1

XML: Extensible Markup Language (XML) 1.0 (Fifth Edition)

JSON: RFC 7159 The JavaScript Object Notation (JSON) Data Interchange Format

1. HTTP Request Format

The API calls are made by making HTTP queries to the ETport or W2-E4 device's IP address on port 80. Either of two HTTP methods, GET and POST, can be used to make the request. GET requests are convenient because they can be tested by simply entering the appropriate URL into a web browser, however, the limitations in the URL syntax prevent some of the more complex API calls from being made; HTTP POSTs must be used in those cases. When using HTTP GET, the API call is embedded into the URL of the request. When using POST, the API call is sent as the body of the request in XML format. The following is an example of the `clock` API call using each of the two methods.

GET method

```
GET /api/clock HTTP/1.1
Accept: application/xml
Connection: close
```

POST method

```
POST /api HTTP/1.1
Accept: application/xml
Connection: close
Content-Type: application/xml
Content-Length: 48

<?xml version="1.0" encoding="UTF-8"?>
<clock/>
```

The `clock` API call is encoded as the path `/api/clock` when using the GET method. When using the POST method, all requests use the generic `/api` path, and the API call is encoded as the empty XML element `<clock/>`.

Using both methods, the Accept header can be set to either "application/xml", "application/json", or "text/plain", according to the desired output type. If the Accept header is not provided, XML will be returned. "text/xml" is supported as an alias to "application/xml". Using the POST method, the Content-Type header must be set to "application/xml" (or "text/xml") and the Content-Length header must be set.

The Connection header may be set to "close" or "keep-alive". Setting it to "close" will cause the server to close the connection after responding, while "keep-alive" will keep the connection open for subsequent requests; see RFC 2616 Hypertext Transfer Protocol – HTTP 1.1 for details.

Note that newlines in HTTP are represented by the byte 13 (hexadecimal 0D) followed by the byte 10 (hexadecimal 0A), sometimes written as "`\r\n`" or "`CR LF`".

The GET method can be easily tested without special software simply by using a web browser; for example, if the IP address of the ETport or W2-E4 device is 192.168.1.10, simply entering the URL `http://192.168.1.10/api/clock` into a browser (such as Mozilla Firefox, Google Chrome, Safari, or Microsoft Edge) to receive the response.

Both of the above methods will return the same result, in XML format. An example response is shown below:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Server: ETPWS v2.31
Transfer-Encoding: chunked
Connection: close

85
<?xml version="1.0" encoding="UTF-8"?>
<result>Thu Mar 07 16:21:04 UTC 2019</result>
0
```

The device may use "chunked" transfer encoding in the response, as in the example above, or may include a Content-Length header, at its option. When using the chunked transfer encoding, the response is broken up into two or more chunks, which are preceded by the number of bytes in each, with the final chunk containing zero bytes. For details on chunked transfer encoding, see RFC 2616 Hypertext Transfer Protocol – HTTP 1.1.

HTTP requests may not be longer than 1 KB in size (though the responses may be of any length).

1.1. Status Codes

A successful API call will return the HTTP response code 200. Other codes are used when an error occurs as follows:

400: If the request could not be parsed, the Content-Type is unsupported, a required parameter was missing, or the value of a parameter was invalid.

401: If a password-protected API call was used without supplying an HTTP Basic Authentication header, or the credentials supplied were incorrect.

404: If no API call was found by the given name.

405: If the wrong method was used (GET or POST) for the URL that was used (i.e., using GET on /api).

406: If the Accept header was invalid or impossible for the given API call.

411: If the Content-Length header was not provided for an XML POST.

413: If the HTTP body was too large (greater than 1KB).

500: If an unexpected server error occurred.

504: If the API call required sending a Modbus message, and the Modbus device did not respond.

2. Template Format

All API calls that are not password-protected can be included in template files for use with the Web Posting feature on ETport and W2-E4 devices. For details on the Web Posting feature in general, see the user manual for those devices.

Templates may contain arbitrary text, allowing data to be posted in any text-based format (such as XML, JSON, or a custom format). API calls are "escaped" by inserting a dollar sign character, "$", followed by the API call and any parameters it requires in brackets (a literal dollar sign may be inserted by using two consecutive dollar signs, "$$").

For example, the following template file contains the API call `clock`:

```
The device's current time is $clock()
```

When this template is posted, an HTTP POST will be sent to the configured web server, such as the following:

```
POST /example/path HTTP/1.1
Content-Length: 57
Connection: close

The device's current time is Thu Mar 07 16:21:04 UTC 2019
```

The ability to insert arbitrary text into the template allows many text-based formats or protocols to be used. For example, if the web server requires POSTs be in the JSON format, the template could be written as follows:

```
{ "current_time": "$clock()" }
```

This will result in the following JSON POST:

```
POST /example/path HTTP/1.1
Content-Type: application/json
Content-Length: 50
Connection: close

{ "current_time": "Thu Mar 07 16:21:04 UTC 2019" }
```

By default, API calls in template files are replaced with text formatted as plain-text. Alternatively, results can be formatted in XML or JSON format, by setting the optional named parameter "format" to "xml" or "json", respectively. See the following section for details on passing parameters. Note that if the web server requires that headers such as Content-Type be set, as in the above example, those can be added in the device's settings (see the user manual on that device for details).

Unlike with HTTP-based API calls, there is no restriction on the size of template files.

3. Parameters

Some API calls accept or require additional parameters to be sent along with the call. Parameters can be named or unnamed, and can be set to a value of the types string (for text), boolean (for true or false), or number (such as 10, 1.2, or 0x2F). For example, the above `clock` API call accepts the optional named parameter `style`, allowing the user to choose between multiple formats. The following example demonstrates the name parameters style set to the string value "ISO8601" (the name of an ISO-standard time format) are passed using the HTTP GET method, the HTTP POST method, and in a template.

HTTP GET method

```
GET /api/clock?style=ISO8601 HTTP/1.1
Accept: application/xml
Connection: close
```

HTTP POST method

```
POST /api HTTP/1.1
Accept: application/xml
Connection: close
Content-Type: application/xml
Content-Length: 64

<?xml version="1.0" encoding="UTF-8"?>
<clock style="ISO8601"/>
```

Template

```
$clock(style = "ISO8601")
```

Example result (HTTP GET or HTTP POST)

```
HTTP/1.1 200 OK
Content-Type: application/xml
Server: ETPWS v2.31
Transfer-Encoding: chunked
Connection: close

74
<?xml version="1.0" encoding="UTF-8"?>
<result>2019-03-07T18:05Z</result>
0
```

Example web post (from template)

```
POST /example/path HTTP/1.1
Content-Length: 17
Connection: close

2019-03-07T18:05Z
```

With the HTTP GET method, parameters are listed following a "?" character; subsequent parameters are separated by a "&" character. With the HTTP POST method, parameters are given as XML attributes. In a template, parameters are given inside the brackets (separated by commas, if there is more than one).

With the HTTP POST method, all parameters must be closed in quotation marks. In a template, only string parameters are enclosed in quotation marks. Quotation marks are not used with the HTTP GET method.

Some parameters are unnamed. For example, in the following example, the `int` API call is used to read a 32-bit integer register from a Modbus device at address 1. The register address, hexadecimal number 0x511, is an unnamed parameter.

HTTP GET method

```
GET /api/int?0x511&mb_address=1 HTTP/1.1
Accept: application/xml
Connection: close
```

HTTP POST method

```
POST /api HTTP/1.1
Accept: application/xml
Connection: close
Content-Type: application/xml
Content-Length: 64

<?xml version="1.0" encoding="UTF-8"?>
<int mb_address="1">0x511</int>
```

Template

```
$int(0x511, mb_address=1)
```

Example result (HTTP GET or HTTP POST)

```
HTTP/1.1 200 OK
Content-Type: application/xml
Server: ETPWS v2.31
Transfer-Encoding: chunked
Connection: close

63
<?xml version="1.0" encoding="UTF-8"?>
<result>1234567</result>
0
```

Example web post (from template)

```
POST /example/path HTTP/1.1
Content-Length: 7
Connection: close

1234567
```

With the HTTP GET method, parameters after the first are separated from the previous characters using a "&" character. With the HTTP POST method, unnamed parameters are specified as the text content of the XML element representing the API call.

(Note that, on the W2-E4, the mb_address parameter is ignored and may be excluded).

The following example shows multiple unnamed parameters given to the add API call, used to add the two numbers 10 and 5 together.

HTTP GET method

```
GET /api/add?10&5 HTTP/1.1
Accept: application/xml
Connection: close
```

HTTP POST method

```
POST /api HTTP/1.1
Accept: application/xml
Connection: close
Content-Type: application/xml
Content-Length: 64

<?xml version="1.0" encoding="UTF-8"?>
<add type="list">
   <item>10</item>
   <item>5</item>
</add>
```

Template

```
$add(10, 5)
```

Example result (HTTP GET or HTTP POST)

```
HTTP/1.1 200 OK
Content-Type: application/xml
Server: ETPWS v2.31
Transfer-Encoding: chunked
Connection: close

65
<?xml version="1.0" encoding="UTF-8"?>
<result>15.000000</result>
0
```

Example web post (from template)

```
POST /example/path HTTP/1.1
Content-Length: 9
Connection: close

15.000000
```

4. Mathematical Operations on Results (templates only)

Four mathematical API calls, add, subtract, multiply, and divide, exist which can be used to do math on the results of other API calls. The mathematical operations on results of other calls may **only** be done from templates.

For example, to add the values of two 32-bit registers at addresses 0x120 and 0x122 together, the following syntax can be used:

Template

```
Value 1: $int(0x120, mb_address=1)
Value 2: $int(0x122, mb_address=1)
Sum:     $add($int(0x120, mb_address=1), $int(0x122, mb_address=1))
```

Example web post

```
POST /example/path HTTP/1.1
Content-Length: 46
Connection: close

Value 1: 50
Value 2: 100
Sum:     150.000000
```

These mathematical operations can in turn be combined with other mathematical operations to create more complex operations. For example, to POST the average of three 32-bit registers, 0x120, 0x122, and 0x124, the following template could be used:

Template

```
Value 1: $int(0x120, mb_address=1)
Value 2: $int(0x122, mb_address=1)
Value 3: $int(0x124, mb_address=1)
Average: $divide($add($int(0x120, mb_address=1), $int(0x122, mb_address=1), $int(0x124, mb_address=1)), 3)
```

Example web post

```
POST /example/path HTTP/1.1
Content-Length: 60
Connection: close

Value 1: 50
Value 2: 100
Value 3: 200
Average: 116.666667
```

5. Output Formats

API calls can produce results in three different formats: XML, JSON, and plain-text. The default output format for HTTP requests is XML, while for templates it is plain-text. The output format can be selected either by supplying the named parameter "format" to an API call (with the value of "xml", "json", or "text"), or by supplying a value for the Accept header in HTTP requests ("application/xml", "application/json", or "text/plain" are accepted).

When making API calls via HTTP requests, the results are additionally wrapped in an XML element (preceeded by a <?xml?> tag), or a JSON object, as appropriate, so that the result is a valid XML or JSON document. Results from templates are not wrapped in this way, so that they can be included into XML or JSON documents as fragments. For example, rather than returning the JSON string `{ result: "a string value" }`, an API call made in a template will simply return `"a string value"`. It is up to the user to design the template to conform with the desired output format.

The following example demonstrates the results of an `ls` API call using HTTP, which lists the contents of a directory in the filesystem, in each of the available formats.

Request 1 (HTTP GET)

```
GET /api/ls HTTP/1.1
Accept: application/xml
Connection: close
```

Example Result 1 (XML)

```
HTTP/1.1 200 OK
Content-Type: application/xml
Server: ETPWS v2.31
Transfer-Encoding: chunked
Connection: close

386
<?xml version="1.0"
encoding="UTF-8"?>
<directory path="/spiffs">
  <entries type="list">
     <entry size="190">
       registers.json
     </entry>
     <entry size="425">
       dummydir/template.json
     </entry>
     <entry size="3297">
       default_template.json
     </entry>
     <entry size="3310">
       cloud.json
     </entry>
  </entries>
</directory>
0
```

Request 2 (HTTP GET)

```
GET /api/ls HTTP/1.1
Accept: application/json
Connection: close
```

Example Result 2 (JSON)

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: ETPWS v2.31
Transfer-Encoding: chunked
Connection: close

395
{
  "directory": {
    "path": "/spiffs",
    "entries": [
      {
        "size": 190,
        "value": "registers.json"
      },
      {
        "size": 425,
        "value": "dummydir/template.json"
      },
      {
        "size": 3297,
        "value": "default_template.json"
      },
      {
        "size": 3310,
        "value": "cloud.json"
      }
    ]
  }
}
0
```

Request 3 (HTTP GET)

```
GET /api/ls HTTP/1.1
Accept: text/plain
Connection: close
```

Example Result 3 (plain text)

```
HTTP/1.1 200 OK
Content-Type: text/plain
Server: ETPWS v2.31
Transfer-Encoding: chunked
Connection: close

280
path: /spiffs
entries:
   entry:
      size: 190
      value: registers.json,
   entry:
      size: 425
      value:  dummydir/template.json,
   entry:
      size: 3297
      value: default_template.json,
   entry:
      size: 3310
      value: cloud.json
0
```

The following example shows the same API call, `ls`, when used from a template file to make a web post. The optional named parameter `pretty_print=2` is also used to add newlines and additional spacing for readability. If the format is left unspecified, template API calls default to plain-text.

Template 1

```
$ls(
   format="xml",
   pretty_print=2
)
```

Template 2

```
$ls(
   format="json",
   pretty_print=2
)
```

Template 3

```
$ls(
   format="text",
   pretty_print=2
)
```

Example post 1 (XML)

```
POST /example/path HTTP/1.1
Content-Length: 332
Connection: close

<directory path="/spiffs">
  <entries type="list">
    <entry size="190">
      registers.json
    </entry>
    <entry size="425">
      dummydir/template.json
    </entry>
    <entry size="3297">
      default_template.json
    </entry>
    <entry size="3310">
      cloud.json
    </entry>
  </entries>
</directory>
```

Example post 2 (JSON)

```
POST /example/path HTTP/1.1
Content-Length: 377
Connection: close

{
  "path": "/spiffs",
  "entries": [
    {
      "size": 190,
      "value": "registers.json"
    },
    {
      "size": 425,
      "value": "dummydir/template.json"
    },
    {
      "size": 3297,
      "value": "default_template.json"
    },
    {
      "size": 3310,
      "value": "cloud.json"
    }
  ]
}
```

Example post 3 (plain text)

```
POST /example/path HTTP/1.1
Content-Length: 260
Connection: close

path: /spiffs
entries:
  entry:
    size: 190
    value: registers.json,
  entry:
    size: 425
    value: dummydir/template.json,
  entry:
    size: 3297
    value: default_template.json,
  entry:
    size: 3310
    value: cloud.json
```

6. Password-Protected API Calls (HTTP only)

Some API calls are password protected. These calls can not be used in a template file. When using these API calls over HTTP, an Authentication header must be supplied. The username is always "admin"; the password is the value specified in the setting `admin_pwd` (see section 9).

The value of the Authentication header must be the string "Basic" followed by the base-64 encoding of the username and password separated by a colon. See RFC 2616 Hypertext Transfer Protocol – HTTP 1.1 for details on this header value.

7. Combining Multiple API Calls (HTTP only)

Multiple API calls can be combined into a single request, for convenience and performance reasons. Due to limitations in HTTP URLs, this can only be done using HTTP POSTs with XML.

This is accomplished using the special API calls `multi` and `multis`. The `multi` API call does not require a password, and can only include other API calls that do not require a password. The `multis` API call requires a password, and can include any other API calls.

The following is an example which combines the `clock`, `uptime`, and `timestamp` API calls into a single HTTP request.

Request (HTTP POST):

```
POST /api HTTP/1.1
Accept: application/xml
Connection: close
Content-Type: application/xml
Content-Length: 113

<?xml version="1.0" encoding="UTF-8"?>
<multi type="list">
   <clock/>
   <uptime/>
   <timestamp/>
</multi>
```

Result:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Server: ETPWS v2.31
Transfer-Encoding: chunked
Connection: close

302
<?xml version="1.0" encoding="UTF-8"?>
<responses type="list">
    <response>
        <result>Fri Mar 08 21:21:45 UTC 2019</result>
    </response>
    <response>
        <result>5048243</result>
    </response>
    <response>
        <result>3761068905</result>
    </response>
</responses>
0
```

8. API Call Listing

This section lists the API calls available on ETport and W2-E4 devices. Parameters are given in brackets, similar to how they appear in template files. Optional parameters are enclosed in square brackets. The ellipses notation "…" indicates that an arbitrary number of arguments may be supplied. Named parameters are always optional and are listed in the format "[name=value]". Unnamed arguments always appear outside of square brackets.

8.1. Special Parameters

Some named arguments are special, able to be used on any API call, or have special rules regarding their use.

8.1.1. [format = f]

This named parameter may be used on any API call. It can be set to the value "xml" (the default), "json", or "text", to control the output format of the result. See section 5 for details on different output formats.

8.1.2. [pretty_print = p]

This named parameter may be used on any API call. If this parameter is included, the results of the API call will be formatted in a way that is easier for humans to read, using newlines and spacing. It is set to an integer between 0 and 10, indicating the number of spaces which are used as indentation for child elements.

8.1.3. [mb_address = a]

This special parameter is mandatory on all Modbus-related API calls (see section 8.2) on ETport devices, but is not used on W2-E4 devices. This is because ETport devices may be connected to multiple Modbus devices, so the address must be indicated to specify which device to communicate with.

Within template files, the mb_address parameter can be given using special syntax, as shown below:

```
$10.short(0x510)
```

This is equivalent to:

```
$short(0x510, mb_address = 10)
```

When using a per-device template, the special value $this may be used in place of a Modbus address (only using the special syntax) to indicate the device that is currently being queried. This allows the same template to be applied to multiple devices, each creating a separate post. For example, if the ETport device is configured for per-device templates with two devices at Modbus address 1 and 2 (see the post_addr_* setting in the last section), the following template will generate two separate web posts, one for each device:

Example Template

```
The device at address $this has uptime $this.uint(0x515) seconds.
```

Example Web Post 1

```
The device at address 1 has uptime 62 seconds.
```

Example Web Post 2

```
The device at address 2 has uptime 1351 seconds.
```

See the ETport manual for a detailed explanation of global and per-device templates.

8.2. Modbus-related API Calls

Each of the following API calls accept a register address parameter. The value of *register_address* must be a value between 0 and 65,535, and may be given in either decimal, hexadecimal, binary, or octal. Note that the first address is 0, not 1, and that "Modicon-style" addresses must be converted to "offset-style" addresses to be used (i.e., the Modicon-style address 40003 is equivalent to the offset-style address 2).

short(register_address [, count=n] [, fc=c])
Returns the 2-byte integer value of the given Modbus register *register_address*. If *count* is given, *n* results will be returned. By default, the Modbus function code 3 will be used; to use function code 4, set the *fc* parameter to 4.

int(register_address [, count=n])
Returns the 4-byte integer value of the given Modbus register register_address. If If *count* is given, *n* results will be returned. By default, the Modbus function code 3 will be used; to use function code 4, set the *fc* parameter to 4.

float(register_address [, count=n], [precision=n])
Returns the 4-byte floating point value of the given Modbus register register_address. If *count* is given, *n* results will be returned. If *precision* is given, the result will be rounded to *p* decimal places (from 0 to 10, default 6). By default, the Modbus function code 3 will be used; to use function code 4, set the *fc* parameter to 4.

ushort(register_address [, count=n])
Returns the 2-byte unsigned value of the given Modbus register register_address. If *count* is given, *n* results will be returned. By default, the Modbus function code 3 will be used; to use function code 4, set the *fc* parameter to 4.

uint(register_address [, count=n])
Returns the 4-byte unsigned value of the given Modbus register register_address. If *count* is given, *n* results will be returned. By default, the Modbus function code 3 will be used; to use function code 4, set the *fc* parameter to 4.

fixedshort(register_address, decimal_point [, count=n])
Returns the 2-byte fixed-point value of the given Modbus register register_address with the decimal place at the given point. For example, if the value of register 1 is 100, fixedshort(1, 2) will return 1.00. If *count* is given, *n* results will be returned. By default, the Modbus function code 3 will be used; to use function code 4, set the *fc* parameter to 4.

slave-id([output=o])

Returns the device's slave ID code or string (as reported by Modbus function 17 Report Slave ID). If *output* is not given, the slave ID code, run indicator flag, and text will be returned. Output may be one of "code", "run-indicator", or "text" to only return the indicated value.

string(register_address, length)
Returns the string value of the given Modbus register register_address with the given maximum *length*. The higher-order byte and the lower-order byte of each register are interpreted as individual ASCII characters, displayed sequentially. By default, the Modbus function code 3 will be used; to use function code 4, set the *fc* parameter to 4.

rdi(code, object [, output = o])
Executes a Read Device Identification Modbus query, used for reading various constants ("objects") to identify the device. If *output* is given, it determines the format of the result, which may be "list" (which outputs all results as a list), "value" (which outputs only the first result as an individual value), or "full" (the default, similar to list, except including additional technical details about the RDI request itself, such as the conformity level). See the Modbus Application Protocol Specification for details on the Read Device Information Modbus query.

ushort-write(register_address, value1 [, value2] [, …])
Writes one or more two-byte register values to a register or block of registers, using the Write Multiple Holding Registers Modbus function. Each value must be between 0 and 65,535 inclusive.

block(register_address, length)
Pre-loads a Modbus block into a cache, making subsequent smaller queries to the register_addresses in that block very fast. This function is simply removed from the template, rather than being replaced with anything. This API call has no effect when used in HTTP requests. By default, the Modbus function code 3 will be used; to use function code 4, set the *fc* parameter to 4.

mb(byte1[, byte2][, …])
Send an arbitrary modbus query byte-by-byte and receive the response (including exception responses). Any number of bytes can be passed to this function. This API call is password protected.

8.3. Mathematical API Calls

add(a, b[, c][, …][, precision=p])
Adds an arbitrary number of arguments together. If *precision* is given, the result will be rounded to *p* decimal places (from 0 to 10, default 6).

subtract(a, b[, c][, …][, precision=p])
Subtracts an arbitrary number of arguments from the first argument, *a*. If *precision* is given, the result will be rounded to *p* decimal places (from 0 to 10, default 6).

multiply(a, b[, c][, …][, precision=p])
Multiplies an arbitrary number of arguments together. If *precision* is given, the result will be rounded to *p* decimal places (from 0 to 10, default 6).

divide(a, b[, c][, …][, precision=p])
Divides the first argument by an arbitrary number of arguments. If *precision* is given, the result will be rounded to *p* decimal places (from 0 to 10, default 6).

8.4. Filesystem-related API Calls

ls([directory])
Lists the contents of the working directory, or of the supplied directory.

mv(current_filename, new_filename)
Moves (or renames) a file specified by *current_filename* to *new_filename*. This API call is password protected.

rm(filename)
Deletes a file specified by *filename* from the filesystem. This API call is password protected.

put(filename, contents)
Creates or overwrites a file specified by *filename* with the contents given by the string *contents*. This API call is password protected.

cat(filename)
Returns the contents of the file specified by *filename*.

get-filesystem()
Returns the number of bytes in use on the filesystem, and the total number of bytes available in the filesystem.

8.5. Settings-related API Calls

get([name])
Returns the value of the setting named *name*, if *name* is supplied, or the value of all settings, if *name* is not supplied.

set(name, value=v)
Sets the value of the setting named *name* to the value *v*, if *v* is valid for this setting. If it is invalid, an error message will be returned indicating the valid range or restrictions for this setting. This API call is password protected.

reset(name)
Resets the setting named *name* to its default value. This API call is password protected.

reset-all(name)
Resets all settings to their default value. This API call is password protected.

8.6. Miscellaneous API Calls

rssi()
Returns the received signal strength indicator of the connected Wi-Fi network, in decibels. If Wi-Fi is not currently connected, 0 is returned.

clock([style=s])
Returns the current time according to the device, as a string of text. If *style* is given, it will be formatted according to style *s*. Available styles are as follows:

Style	**Example**
UNIX (default)	Thu Mar 07 16:21:04 UTC 2019
ISO8601	2019-03-07T16:21Z
Legacy	2019/03/07 16:21:04 UTC

timestamp()
Returns the number of seconds since January 1st, 1900 at 00:00:00 UTC.

uptime()
Returns the number of milliseconds that have passed since the device was last turned on or power-cycled.

gateway-device()
Returns the part name of this device (the gateway).

gateway-name()
Returns the name of this device (the gateway).

gateway-version()
Returns the version number of this device (the gateway).

live()
Returns false if this post is buffered, true otherwise. Always returns true if used outside of a template.

mac()
Returns the MAC address of this device (depending on which interface is being used).

cloud-id()
Returns the device's Cloud ID, used when adding the device to an Elkor Cloud account. For this device, this is simply the Ethernet MAC address (even if Wi-Fi is being used).

reboot()
Reboots the device. This API call is password protected.

post-now()
Triggers an immediate web post (rather than waiting for the posting interval to elapse), if posting is enabled and configured.

clear-buffer()
Deletes any buffered posts from system flash memory.

diagnostics()
Returns task and memory statistics for debugging purposes.

connections()
Returns a list of active connections to the device, for debugging purposes. This API call is expensive and can adversely affect the performance of the device, so take care to avoid making repeated calls frequently.

help()
Returns a list of the API calls and a brief description.

find()
Returns a list of IP address and configuration information, including the device, name, firmware version and release date, ip address, subnet mask, default gateway, DNS servers, and Wi-Fi SSID.

get-firmware-configuration()
Gets information about the device's firmware, including the version number, release date, and operating system version.

get-status()
Gets status information for the device, including the uptime, Modbus activity, posting status, and all of the information returned by the `find` API call.

nofail()
When used in a template, this API call allows individual API calls to fail without causing the entire template to fail. Failed API calls will output an error string describing the failure, rather than their usual output value. This API call has no effect when used outside of a template file.

blink()
Causes one of the device's LEDs to blink in a specific pattern, allowing it to be visually identified in the presence of multiple similar devices. The blink pattern lasts for 4 seconds. For W2-E4 devices, the LED is the right orange LED on the Ethernet jack. For ETport devices, the LED is the green "Status" LED. The pattern was chosen to avoid resembling that of normal activity. The blink pattern can be described as follows, with orange indicating "on" and black indicating "off":

9. Setting Listing

This section lists the available settings on ETport and W2-E4 devices. These can be read or written to using the `get` and `put` API calls (described in the previous section).

Setting Name	Default Value	R/RW	Range/Restrictions	Description
commissioned	1	R	0-1	Set at the factory. When set to 0, all settings may be written regardless of their R/RW field in this table.
mb1_baud	9600	R (W2-E4); RW (ETport)	9600-460,800; multiple of 9600	Baud rate of the Modbus Port 1.
mb1_parity	0	R (W2-E4); RW (ETport)	0-2	Parity mode of Modbus Port 1. 0 = no parity; 1 = odd parity; 2 = even parity.
mb1_stop	1	R (W2-E4); RW (ETport)	1-2	Number of stop bits used on Modbus Port 1.
mb2_baud	9600	RW	9600-460,800; multiple of 9600	This setting currently has no effect on this device.
mb2_parity	0	RW	0-2	This setting currently has no effect on this device.
mb2_stop	1	RW	1-2	This setting currently has no effect on this device.
tcp_idle_time	7200	RW	1-1,000,000	Number of seconds a TCP connection may be idle before being forcibly closed by the server to free resources.
mbtcp_port	502	RW	1-65,535	TCP port for the Modbus TCP server to listen on.
http_port	80	RW	1-65,535	TCP port for the HTTP server to listen on.
https_port	443	RW	1-65,535	This setting currently has no effect.
tn_port	23	RW	1-65,535	TCP port for the telnet server to listen on.
api_udp_port	30139	RW	1-65,535	UDP port for the API UDP server to listen on.
api_tcp_port	30138	RW	1-65,535	This setting currently has no effect on this device.
post_en	0	RW	0-1	Enable or disable web posting. 0 = disabled; 1 = enabled.
post_seconds	60	RW	0-999,999,999	The interval between making web posts.
post_auth_en	0	RW	0-1	Enable or disable HTTP Basic authentication on web posts. 0 = disabled; 1 = enabled.
post_timeout_s	60	RW	0-1,000,000	Maximum seconds to wait for a response when making a web post to a server before giving up.
post_buf_en	1	RW	0-1	Enable or disabled buffering to the filesystem for failed web posts. 0 = disabled; 1 = enabled.
post_buf_limit	60	RW	0-100	Percentage of the available filesystem space that may be used by buffered posts. Any buffered posts in excess of this limit will be discarded. It is recommended to leave some amount of space free to allow for firmware updates (which are typically about 1MB in size).
ntp_en	1	RW	0-1	Enable or disable the Network Time Protocol client for automatic time/date acquisition. 0 = disabled; 1 = enabled.
ntp_listen_port	123	RW	1-65,535	UDP port to listen for NTP responses on.
ntp_port_0	123	RW	1-65,535	UDP port of the first server to send NTP requests to.

ntp_port_1	123	RW	1-65,535	UDP port of the second (backup) server to send NTP requests to.	
ntp_port_2	123	RW	1-65,535	UDP port of the third (backup) server to send NTP requests to.	
ntp_rate_m	1440	RW	30-65,535	Number of minutes between making NTP requests.	
ntp_timeout_ms	5000	RW	10-65,535	Number of milliseconds to wait for an NTP response before giving up.	
ntp_retry_ms	2000	RW	10-65,535	Number of milliseconds to wait before retrying an NTP request after the previous one failed.	
eth_static_ip	0x0164a8c0	RW	0-4,294,967,295	Static IP address for the ethernet interface.	The IP address is encoded as a hexadecimal 32-bit little-endian integer, i.e., 0x01020304 = 4.3.2.1. This setting has no effect unless eth_mode is set to 0. Changing this setting will cause the device to reboot.
eth_static_nm	0x00ffffff	RW	0-4,294,967,295	Static subnet mask for the ethernet interface.	
eth_static_gw	0xfe64a8c0	RW	0-4,294,967,295	Static default gateway for the ethernet interface.	
eth_static_dn0	0xfe64a8c0	RW	0-4,294,967,295	Static DNS server address for the ethernet interface.	
eth_static_dn1	0x00000000	RW	0-4,294,967,295	Static backup DNS server address for the ethernet interface.	
wifi_static_ip	0x0264a8c0	RW	0-4,294,967,295	Static IP address for the Wi-Fi interface.	
wifi_static_nm	0x00ffffff	RW	0-4,294,967,295	Static subnet mask for the Wi-Fi interface.	
wifi_static_gw	0xfe64a8c0	RW	0-4,294,967,295	Static default gateway for the Wi-Fi interface.	
wifi_static_dn0	0xfe64a8c0	RW	0-4,294,967,295	Static DNS server address for the Wi-Fi interface.	
wifi_static_dn1	0x00000000	RW	0-4,294,967,295	Static backup DNS server address for the Wi-Fi interface.	
http_sto_ms	30000	RW	0-4,294,967,295	Send timeout for the HTTP server.	
http_rto_ms	30000	RW	0-4,294,967,295	Receive timeout for the HTTP server.	
auto_reboot_s	86400	RW	0-4,294,967,295	Automatically reboot the device after the given number of seconds. Set to 0 to disable.	
reconnect_s	300	RW	0-4,294,967,295	Number of seconds to wait after being disconnected from all networks before forcing a reconnect attempt.	
telnet_en	0	RW	0-1	Enable or disable the telnet server. 0 = disabled; 1 = enabled.	
tolerate_no_i2c	0	RW	0-1	This setting currently has no effect on this device.	
format	0	RW	0-65,535	Set to 57,005 to format (completely erase) the device's filesystem.	
log_size_limit	1048576	RW	0-4,000,000	The maximum number of bytes that can be consumed by a log file. This setting has no effect unless log_filename is set.	
wifi_chan_start	1	RW	1-14	The first Wi-Fi channel to scan when connecting to a	

				Wi-Fi network. Local regulations may limit the legal values for this setting.
wifi_chan_count	11	RW	1-14; wifi_chan_start + wifi_chan_count – 1 <= 14.	The number of Wi-Fi channels to scan. Local regulations may limit the legal values for this setting.
wifi_max_tx_pwr	100	RW	100-10000	The transmit power of the Wi-Fi client. Local regulations may limit the legal values for this setting.
attached_devs	0 (ETport); 9 (W2-E4);	R	0-25	Bitwise-OR of device codes that are embedded into this device's hardware, used internally for determining whether a firmware file is valid for this device or not. STANDALONE = 0; WATTSONMKII = 1; ISPY = 2; WATTSONMKII_ISPY = 3; ETNET = 4; WATTSONMKII_ETNET = 5; ISPY_ETNET = 6; WATTSONMKII_ISPY_ETNET = 7; IDLM = 8; WATTSONMKII_IDLM = 9; ECM = 16; WATTSONMKII_ECM = 17; ECM_ETNET = 20; ECM_WATTSONMKII_ETNET = 21; ECM_IDLM = 24; WATTSONMKII_ECM_IDLM = 25
etport_variant	0	R	0-2	Indicates the type of this device. ETPORT = 0; ETPORT_W2: 1; ETPORT_MCM = 2
eth_mode	1	RW	0-1	Set the ethernet IP mode to either use a static IP address, or to get one automatically from the router with the DHCP protocol. If set to use a static IP address, all eth_static_* must be configured. STATIC = 0; DHCP = 1
wifi_mode	1	RW	0-1	Set the wifi IP mode to either use a static IP address, or to get one automatically from the router with the DHCP protocol. If set to use a static IP address, all wifi_static_* must be configured. STATIC = 0; DHCP = 1
post_time_unit	1	RW	0-2	Stores whether the web posting interval has been chosen in seconds, minutes, or hours on the web interface. SECONDS = 0; MINUTES = 1; HOURS = 2
name	ETport	RW	32-char string	User-setting name for this device.
wifi_ssid		RW	33-char string	SSID of the Wi-Fi network to connect to. Note that this device will only connect to a Wi-Fi network if it is not connected to a wired ethernet network.
wifi_pwd		RW	65-char string	Password/key of the Wi-Fi network to connect to.
admin_pwd	admin	RW	16-char string	Administrator password used for password-protected API calls and web-interface pages.
post_url		RW	256-char string	The URL to send web posts to. This setting has no effect unless post_en is set to 1.
post_auth_user		RW	32-char string	The username to use for HTTP Basic authentication when making when making web posts. This setting has no effect unless post_auth_en is set to 1.
post_auth_pass	<not shown>	RW	32-char string	The password to use for HTTP Basic authentication when making when making web posts. This setting has no effect unless post_auth_en is set to 1.
ntp_server_0	time.nist.gov	RW	64-char string	The IP addresses or DNS names of NTP servers to try sending NTP requests to. Up to three may be specified, which are tried in rotating order until one responds. This setting has no effect unless ntp_en is set to 1.
ntp_server_1	pool.ntp.org			
ntp_server_2	pool.time.org			

post_file_0	/spiffs/template.json	RW	64-char string	Filenames of template files to use when web posting. Up to 10 different template files may be specified. Each template file will by processed in order and a web post will be made to the server specified in the post_url setting. These settings have no effect unless post_en is set to 1.
post_file_1		RW	64-char string	
post_file_2				
post_file_3				
post_file_4				
post_file_5				
post_file_6				
post_file_7				
post_file_8				
post_file_9				
post_method	POST	RW	32-char string	The HTTP method to use when posting. Generally, this should be set to POST (the default), GET, or PUT, according to the server's requirements, but any string may be entered for use with custom web servers.
post_headers		RW	256-char string; valid header format	Any additional headers to send with each web post. If not empty, this setting must contain one or more lines containing a ':' and ending with the bytes 13, and 10 (sometimes referred to as "CR LF" or "\r\n"). Header names and values must be ASCII bytes between 32 and 126. An example is as follows: My-Custom-Header: a value\r\n My-Custom-Header-2: another value\r\n
post_ct		RW	32-char string	The value to use for the Content-Type header for the web post. If empty, no Content-Type header will be sent. In general, this should match the format of the template, such as "application/xml" for XML templates, "application/json" for JSON templates, etc.
http_headers		RW	256-char string	Any additional headers for the HTTP server to send when responding to HTTP requests. If not empty, this setting must contain one or more lines containing a ':' and ending with the bytes 13, and 10 (sometimes referred to as "CR LF" or "\r\n"). Header names and values must be ASCII bytes between 32 and 126. An example is as follows: My-Custom-Header: a value\r\n My-Custom-Header-2: another value\r\n
debug_tag_0	MBTCP	RW	32-char string	Tag to enable additional messages to be displayed in the log, for debugging purposes.
debug_tag_1	HTTP	RW	32-char string	Tag to enable additional messages to be displayed in the log, for debugging purposes.
debug_tag_2		RW	32-char string	Tag to enable additional messages to be displayed in the log, for debugging purposes.
debug_tag_3		RW	32-char string	Tag to enable additional messages to be displayed in the log, for debugging purposes.
log_file		RW	32-char string	Filename for a log file. If not empty, a copy of the log will be written to the filesystem at this filename. It is not recommended to enable this permanently, as it has a significant performance penalty, and will eventually wear out the flash memory.
wifi_country	WW	RW	3-char string	Two-character country code representing the local regulations affecting the Wi-Fi channels and transmission power. Defaults to "WW" representing "World-wide" values that should be valid in most

				places.
post_addr_0 post_addr_1 post_addr_2 post_addr_3 post_addr_4 post_addr_5 post_addr_6 post_addr_7 post_addr_8 post_addr_9	0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0000000000 0	RW	32 bytes, each between 0 and 248.	Null-terminated list of Modbus addresses corresponding to each template file. If the first byte of this setting is 00, the corresponding template is a "global template," and all subsequent bytes are ignored. Global templates are processed once each, resulting in a single web post for each template. If the first byte is not 00, the corresponding template file is a per-device template. This template will be processed once for each non-zero byte in the value of this setting until the first 00 byte is reached. This results in separate web posts for each Modbus device.