W&T

www.WuT.de

Reference manual

Programming and protocols

Web-IO Digital 4.0

Web-IO Analog 4.0

Web-Thermometer

Web-Thermo-Hygrometer

Web-Thermo-Hygrobarometer

Release 1.79 04/2026

© 02/2026 by Wiesemann und Theis GmbH

Microsoft, MS-DOS, Windows, Winsock and Visual Basic
are registered trademarks of Microsoft Corporation
Subject to error and alteration:
Since it is possible that we make mistakes, you mustn't use
any of our statements without verification. Please, inform us
of any error or misunderstanding you come about, so we can
identify and eliminate it as soon as possible.

Carry out your work on or with W&T products only to the extent that they are described here and after you have completely
read and understood the manual or guide. We are not liable
for unauthorized repairs or tampering.

Content

Overview 6
Access from your own application 6
Modbus TCP - standardised access 6
TCP/IP-Sockets – Command strings 7
TCP Server 12
UDP Peer 12
Communication between Web-IO and application 13
Querying the outputs 16
Switching the outputs 16
Querying the counters 18
Clearing the counters 19
At hoc reading of inputs, outputs and counters 20
Communication between Web-IO Analogue and Application 20
Querying the IO channels 20
Set current/voltage value of an output 22
Web-IO Climate, VOC and Application 22
Querying the IO channels 22
Query and set date and time 23
Querying date and time 23
Setting the date and time 23
Reading the error memory 24
TCP/IP Sockets - Binary 28
TCP server 28
TCP client 29
UDP peer 29
BINARY – The IO structures 29
Definition of the structures 30
Password protected access 32
Unauthorized access attempt 32
The structure Authrequired 32
Login procedure 32
The structure KeyRequest 33

The structure Key 33
The structure Login 34
The structure AuthOK 34
The structure AuthFail 35
Digital IO-Access with Binary Structures 35
The structure ReadRegister 35
The structure WriteRegister 35
The structure SetBit 36
The structure RegisterRequest 37
The structure RegisterState 37
The structure Send Mode 38
The structure ReadCounter 38
The structure ReadClearCounter 39
The structure Counter 39
The structure ReadAllCounter 40
The Structure ReadClearAllCounter 40
The structure AllCounter 41
The structure ClearCounter 41
The struktur Options 42
IO structures for reading the logger content (#57756) 43
The struktur LOGGER_REQUEST 43
The struktur LOGGER_ NOACCESS 43
The struktur LOGGER_CHECK 44
The struktur LOGGER_DATAREQUEST 45
The struktur LOGGER_DATASET 45
Analog I/O access with binary structures 48
The structure ReadRegister 48
The IO structure AnalogRegisterState 48
The structure AnalogSingleRegister 49
The IO structure Send Mode 50
The IO structure Options 51
Device status via binary structures 52
The structure ReadDiagnosis 52
The structure Diagnosis 52
The structure ClearDiagnosis 53
Device features via binary structure 54
The structure InventoryRequest 54

HTTP-Request 56
Supported requests (commands) 56
An example for use of HTTP requests 61
HTML - structure of a static web page 61
JavaScript and AJAX - Change content dynamically 63
CORS - Cross Origin Resource Sharing 66
Using W&T Tags 68
HTTP requests outside the browser 70
REST - Representational State Transfer 72
JSON 72
Read access 72
Changing access 75
XML 77
Reading access 77
Changing access 78
Text 79
Reading access 79
Modbus TCP - standardized access 80
Modbus TCP communication 80
Function Code 0x01 Read Coils 83
Function Code 0x02 Read Discrete Inputs 85
Function Code 0x03 Read Holding Registers 85
Function Code 0x04 Read Input Registers 87
Function Code 0x05 Write Single Coil 88
Function Code 0x06 Write Single Register 89
Function Code 0x0F Write Multiple Coils 90
Function Code 0x10 Write Multiple Registers 92
Function Code 0x07 Read Exception State 93
Incorrect master requests 94
Modbus address ranges for Web-IO Digital 95
Modbus memory layout 97
Modbus address ranges for Web-IO Analog 99
Modbus address ranges for Web-Thermometer 103
Virtual registers for browser interaction 105

Overview

Access from your own application

In addition to the numerous standard access possibilities, the Web-IO also offers you the option of accessing it from one of your own applications.

This can be done using TCP/IP sockets from the common high-level languages – but it is also possible to use familiar Web technologies such as AJAX or PHO to communicate with the Web-IO.

The Web-IO offers several methods for individual access:

- Command strings ASCII
- Binary structures BINARY
- HTTP-Requests REST
- HTTP-Requests JavaScript/AJAX

Numerous programming examples in various programming languages can be found at http://wut.de in the Web-IO Digital section. Under More information about this product group follow the Tools link and there click on the Programming examples general tab.

Modbus TCP - standardised access

Even though MODBUS-TCP is not usually used as a communication path from within your own application, we would like to provide an overview of the function codes and register addresses used with the Web IO at this point.

TCP/IP-Sockets – Command strings

By exchanging simple command strings you can read the inputs and counters and set the outputs.

Depending on the configuration the Web-IO operates in this mode as a TCP server or as a UDP peer.

The data exchange between application and Web-IO takes place using simple command strings.

- The command strings consist of
- The prefatory sequence: GET /
- The actual command: <command>

- The delimiter: ?
- and one or more parameters: `<parameter1>&<...`

The sequence of the parameters is prescribed and cannot be freely chosen.

All command strings always end with `&`

Command strings are case-sensitive.

No CR LF (0x0d 0x0a) may be appended to the line end.

The reply from the Web-IO consists of an identifier which indicates what is being sent, separated by a semicolon, and the values, which are also separated by a semicolon. At the end a null byte is appended (0x00).

```
<identifier>;<value>[;<value2>;<valuex>]+(0x00)
```

If the Web-IO is configured as a server for TCP sockets, the IP address and device name can be sent before the actual reply by setting the option Prefix header.

```
<ip address>;<device name>;<identifier>;<value>[;<value2>;<valuex>]+(0x00)
```

In addition, the reply is inserted in an HTTP frame and only then sent by setting the option Reply HTTP conformal.

The following list contains an overview of all supported commands with their associated parameters.

Web-IO Digital

Commands	Parameter	Description
`GET /input`**`x`**	`?PW=`**`password`**`&` The administrator or user password must be used instead of `password`. If no password has been assigned, `PW=&` is entered. (applies to all commands!)	Input status request `x` optionally specifies the no. of the input and can be between 0 and 16 depending on the model. The feedback of the Web IO is a string starting with `inputx`; followed by the input status: `ON` = signal at the input and `OFF` = no signal at the input If `x` is omitted completely, the Web IO returns a bit pattern corresponding to the input signals in hexadecimal notation.
`GET /counter`**`x`**	`?PW=`**`password`**`&`	Counter value request `x` can be a value between 0-11 and indicates the input. The feedback from the Web IO is a string beginning with `counterx`;. The counter value of the selected counter is appended in decimal notation.
`GET /counter`	`?PW=`**`password`**`&`	Requesting all counter values The feedback of the Web IO is a string beginning with `counter`;. The counter values are added in decimal notation separated by semicolons.
`GET /outputx`	`?PW=`**`password`**`&`	Output status request x optionally specifies the No. of the output and can be between 0 and 16 depending on the model. The feedback from the Web IO is a string starting with `outputx`; followed by the output status: `ON` = signal at the output and `OFF` = no signal at the output If `x` is omitted completely, the Web IO returns a bit pattern corresponding to the output signals in hexadecimal notation.

Commands	Parameter	Description
GET /outputaccessx	?PW=**password**& [Mask=**XXXX**&] State=**ON**/**OFF**/**YYYY**& [NA=ON&] ON: Output = 1, OFF: Output = 0, TOGGLE: Change of state **XXXX**: Hex value between 0000 and 0FFF according to the bits that are to be set **YYYY**: Hex value between 0000 und 0FFF according to the output bit pattern.	Set one or more outputs x can be a value between 0-11 and specifies the output to be set. The feedback of the Web IO is a string starting with „output;“ followed by a bit pattern corresponding to the output signals in hexadecimal notation. The specification of Mask is optional. If Mask is not sent, the outputaccess command applies to all outputs. With NA=ON it is optionally achieved that no response to the outputaccess command is sent by the Web IO.
GET /counterclear**x**	?PW=**password**& [Set=**value**&] **value**: Counter preset, value between 0 and 2 billion	Sets the counter value of a counter. If the Set parameter is not sent, the default is 0. x can be a value between 0-11 and specifies the input whose counter is to be reset. The response of the Web IO is a string beginning with counterx;. The new count of the selected counter is appended in decimal notation. If x is not specified, all counters are set.
GET /allout	?PW=**password**&	Collective request of the input, output states and all counter values. The Web IO responds with a string of the following structure: input;0xxx;output;0xxx-;counter;n0;n1;n2,... 0xxx corresponds to the status of the inputs or outputs in hexadecimal notation. n0,n1 etc. contain the counter states in decimal notation.
GET /time	?PW=**password**&	Returns the system time of the Web IO. Format: **DD.MM.YYYY,hh:mm:ss** **D**=Day, **M**=Month, **Y**=Year, **h**=Hour, **m**=Minute, **s**=Second
GET /settime	?PW=**password**& time=**DD.MM.YYYY,** **hh:mm:ss**&	Sets the system time of the Web IO to the value passed with time.
GET /diagnosis	?PW=**password**&	Requests the status of the diagnostic memory. The Web IO returns: diagnosis;0000;00000000;00000000;00000000 the four-digit value indicates the number of stored messages. With the three eight-digit hexadecimal values, each set bit represents one of the 92 possible messages.

Commands	Parameter	Description
GET /diagnosis**x**	?PW=**password**&	With x, the index for one of the currently stored messages is specified. The Web IO sends the corresponding message text as a return. x must not be greater than the number of the currently present messages.
GET /diaglist**x**	?PW=**password**&	Returns the messages for the individual message bits (max. 92)
GET /diagclear	?PW=**password**&	Clears message memory
GET /errorclear	?PW=**password**&	Clears load errors and releases the affected outputs.

Web-IO Analog

Commands	Parameter	Description
GET /single**x**	none	Request for the current current or voltage values in mA or V. x optionally indicates the no. of the IO channel and can be 1 or 2. The feedback of the Web-IO is a string that shows the value with three decimal places and unit [N]N.NNN mA or [N] N.NNN V If x is omitted completely, the Web IO returns the values of both channels separated by semicolons.
GET /output**x**	none	gives the same result as GET /single**x** (even if IO channels work as input)
GET /outputaccessx	?PW=**password**& State=**N,NNN**& **N,NNN:** Strom- bzw. Spannungswert der am entsprechenden output eingestellt werden soll	Setting an output x can be 1 or 2 and indicates the output to be set. The feedback of the Web IO is a string in the format [N]N.NNN mA or [N] N.NNN V. and indicates the current value. Please note that the Web IO needs a few ms to set the desired value. Therefore, the value does not correspond to the desired value.
GET /time	?PW=**password**&	Returns the system time of the Web IO. Format: **DD.MM.YYYY,hh:mm:ss** **D**=Day, **M**=Month, **Y**=Year, **h**=Hour, **m**=Minute, **s**=Second
GET /settime	?PW=**password**& time=**DD.MM.YYYY,hh:mm:ss**&	Sets the system time of the Web IO to the value passed with time.

Commands	Parameter	Description
GET /diagnosis	?PW=**password**&	Requests the status of the diagnostic memory. The Web IO returns: diagnosis;0000;00000000; 00000000;00000000 the four-digit value indicates the number of stored messages. With the three eight-digit hexadecimal values, each set bit represents one of the 92 possible messages.
GET /diagnosis**x**	?PW=**password**&	With x, the index for one of the currently stored messages is specified. The Web IO sends the corresponding message text as a return. x must not be greater than the number of the currently present messages.
GET /diaglist**x**	?PW=**password**&	Returns the messages for the individual message bits (max. 92)
GET /diagclear	?PW=**password**&	Clears message memory

Web-IO Klima (Web-Thermometer, ...) und VOC

Commands	Parameter	Description
GET /single**x**	none	Request for the current climate data x optionally indicates the no. of the sensor. The feedback of the Web-thermometer is a string that shows the value with one decimal digit and unit 24,0°C If x is omitted completely, the Web IO returns the values of both channels separated by semicolons.
GET /time	?PW=**password**&	Returns the system time of the Web IO. Format: **DD.MM.YYYY,hh:mm:ss** **D**=Day, **M**=Month, **Y**=Year, **h**=Hour, **m**=Minute, **s**=Second
GET /settime	?PW=**password**& time=**DD.MM.YYYY, hh:mm:ss**&	Sets the system time of the Web IO to the value passed with time.
GET /diagnosis	?PW=**password**&	Requests the status of the diagnostic memory. The Web IO returns: diagnosis;0000;00000000; 00000000;00000000 the four-digit value indicates the number of stored messages. With the three eight-digit hexadecimal values, each set bit represents one of the 92 possible messages.

Commands	Parameter	Description
`GET /diagnosis`**`x`**	`?PW=`**`password`**`&`	With x, the index for one of the currently stored messages is specified. The Web IO sends the corresponding message text as a return. x must not be greater than the number of the currently present messages.
`GET /diaglist`**`x`**	`?PW=`**`password`**`&`	Returns the messages for the individual message bits (max. 92)
`GET /diagclear`	`?PW=`**`password`**`&`	Clears message memory

TCP Server

To access the Web-IO as a TCP server via ASCII sockets, enable TCP ASCII-Sockets under *Communication paths >> Socket API*. Specify which TCP-Port should be used to receive Web-IO connections. The Web-IO can provide up to eight TCP connections at the same time on the specified port (only 7 if UDP-ASCII access is also used), and any additional connection attempt is rejected.

If the Web-IO does not receive a valid command within 30 seconds after the connection is opened, it closes the connection and enables access again. The Web-IO behaves the same way if a faulty or unknown command is received.

To be able to switch the outputs using ASCII sockets, Enable *outputs for ASCII-Sockets* must be checked.

UDP Peer

In contrast to TCP, there is no fixed connection between the Web-IO and the other communication partner when using the UDP protocol. Data are exchanged via datagrams.

To access the Web-IO as a UDP peer via ASCII sockets, enable UDP ASCII-Sockets under *Communication paths >> Socket API*. Specify which UDP-Port should be used for the Web-IO to receive UDP datagrams under Local UDP Port.

In normal situations a 0 should be entered as the Remote UDP Port so that the Web-IO sends reply datagrams to the UDP port from which the request was issued. Alternately a fixed port to which the Web-IO sends all datagrams can be specified under Remote UDP Port.

UDP communication always takes place by polling. This means the application uses the command strings to request the desired values and set the outputs.

To be able to switch the outputs using UDP ASCII sockets, Enable outputs for UDP-Sockets must be checked.

Communication between Web-IO and application

Querying the inputs
The inputs are generally read using polling. This means the client application uses the command strings to request the desired values.

Either the status of one particular input can be queried or all inputs are queried.

To query just one input, add the keyword `input` to the input number. To query all inputs simply skip the indication.

If the Web-IO is password protected, the password is appended with ? as a separator with the statement `PW`

The command string ends with &

If no password has been assigned, the `&` follows directly behind `PW=`

Example for querying Input1 with the password “blue” set:

```
GET  /input1?PW=blue&
```

The Web-IO then returns:

```
input1;ON+(0x00)
```

if a valid signal is sent to the input, or:

```
input1;OFF+(0x00)
```

If no signal is present. All replies from the Web-IO end with a null byte (0x00).

To query all inputs the command is:

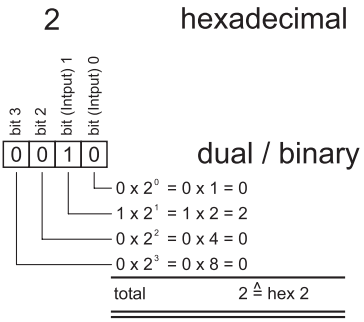
```
GET /input?PW=blue&
```

The Web-IO replies with the keyword input and a one- or four-character hex number. The hex number corresponds to the bit pattern which results from the set or not-set inputs

Example for Web-IO 2xIn,2xOut with Input 1 = ON:

```
input;2+(0x00)
```

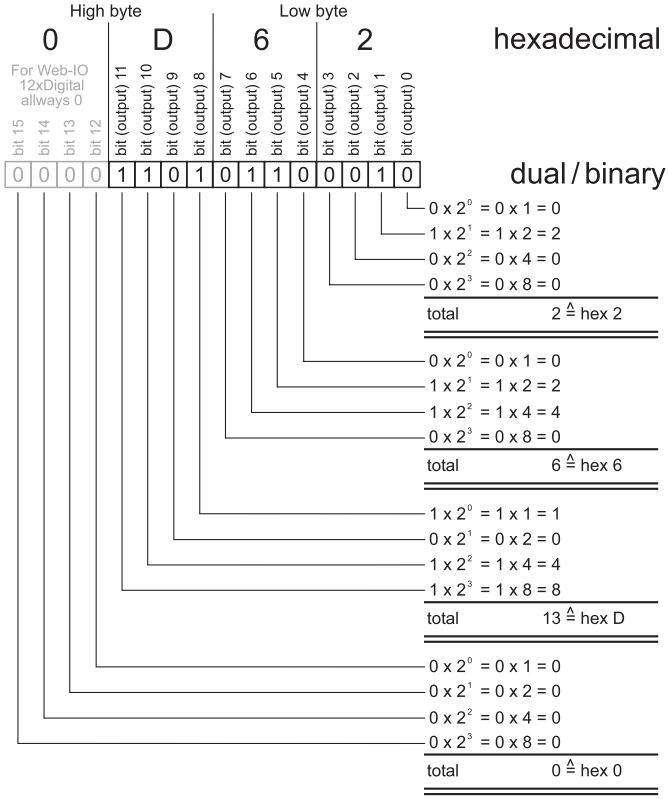
The Value is calculated according to the following calculation:



Example for Web-IO 12xIn, 12xOut with Inputs 1,5,6,8,10 and 11 = ON:

```
input;0D62 +(0x00)
```

The Value is calculated according to the following calculation:



Event-driven messaging (TCP only)

Some applications require that changes to certain outputs are automatically reported on the existing TCP connection.

One reason can be reducing the data load on the network. If the inputs are queried a cyclical intervals, there is a continuous data load. There is also the risk that changes to the inputs between two cycles may not be recognized.

To have the Web-IO automatically send signal changes on the inputs, check the corresponding Input Triggers under *Communication paths >> Socket access*.

The messages from the Web-IO have the same structure as the replies when querying all inputs

Querying the outputs

Read access to the outputs is done in the same manner as querying the inputs, except that the keyword `input` is replaced with `output`

Switching the outputs

To switch the outputs use the command `outputaccess`.

Either a single output, all outputs at the same time, or some of the outputs can be switched.

Switching one output

As with reading, the output is specified. The number of the output to be switched is appended to the keyword `outputaccess`

In addition, the parameter State is used, for which in this case there are three possible values:

- `ON`
- `OFF`
- `TOGGLE` (state change)

Example:

```
GET /outputaccess1?PW=blue&State=ON&
```

switches Output 1 to ON.

Als Antwort sendet das Web-IO z.B.

```
output;0EAC +(0x00)
```

whereby the actually returned value depends on which state the other outputs were in before switching.

	bitwise representation	hexadecimal
Initial state of the outputs:	0000 1110 1010 0100	output; 0EA4
Output 1:	1	State=ON
State after switching:	0000 1110 1010 0110	output; 0EAC

Simultaneous switching of all outputs

To switch all outputs at the same time, no output is specified and the State parameter is used to send the hex number for the desired bit pattern of the outputs.

If for example you wish to switch outputs 1, 6 and 11 to ON and all others to OFF, the command looks as follows:

```
GET /outputaccess?PW=blue&State=0842&
```

in reply the Web-IO sends

```
output;0842 +(0x00)
```

	bitwise representation	hexadecimal
Initial state of the outputs:	0000 1110 1010 0100	output; 0EA4
State:	0000 1000 0100 0010	State=0842
State after switching:	0000 1000 0100 0010	output; 0842

Simultaneous switching of selected outputs

To switch only some of the outputs, the same command structure as for switching all outputs is used. The difference is that the Mask parameter precedes the State parameter. The Mask parameter specifies which outputs should be switched, and State specifies to what state.

For example to switch outputs 1, 6 and 11 to ON and outputs 5 and 10 to OFF, the command looks as follows:

```
GET /outputaccess?PW=blue&Mask=0C62&State=0842&
```

in reply the Web-IO sends

```
output;0AC6 +(0x00)
```

whereby the actually returned value depends on which state the other outputs were in before switching.

	bitwise representation	hexadecimal
Initial state of the outputs:	0000 1110 1010 0100	output; 0EA4
Mask:	0000 1100 0110 0010	Mask=0C62
State:	0000 1000 0100 0010	State=0842
State after switching:	0000 1010 1100 0110	output; 0AC6

Querying the counters

The Web-IO inputs include a counter which is incremented by one with each rising edge, i.e. a change from OFF to ON.

The counters are queried using the command `counter`. As with the inputs and outputs, the values for all counters or just a single counter can be queried.

When a counter state of 2147483648 is reached
it turns over to 0

Querying one counter

The counter number of the input whose counter should be queried is appended to the keyword `counter`.

Example for querying Counter 0:

```
GET /counter0?PW=blue&
```

in reply the Web-IO sends e.g.

```
counter;3974 +(0x00)
```

The counter value is returned in decimal format, separated by a semicolon, behind the keyword `counter`.

Querying all counters

If the `counter` command is sent to the Web-IO without specifying the number of an input, the Web-IO returns the counter states of all inputs separated by semicolons.

Example for querying all counters of a Web-IO with 12 inputs:

```
GET /counter?PW=blue&
```

in reply the Web-IO sends e.g.

```
counter;3974;453;99;0;0;984;712;4;334;1076;0;6543 +(0x00)
```

The counter value is returned in decimal format and semicolon separated behind the keyword `counter`.

Clearing the counters

The command `counterclear` can be used to clear a specified or all counters to 0 or if needed to set it/them to another value.

Clearing one counter

Example for clearing Counter 0:

```
GET /counterclear0?PW=blue&
```

in reply the Web-IO sends

```
counter0;0 +(0x00)
```

Presetting one counter

To preset a counter to a particular value, the optional parameter `Set=<value>&` is appended.

Example for setting Counter 0 to 12345:

```
GET /counterclear0?PW=blue&Set=12345&
```

in reply the Web-IO sends

```
counter0;12345 +(0x00)
```

Clearing all counters

Example for clearing all counters for a Web-IO with 12 inputs:

```
GET /counterclear?PW=blue&
```

in reply the Web-IO sends

```
counter0;0;0;0;0;0;0;0;0;0;0;0 +(0x00)
```

Presetting all counters

To preset all counters to a particular value, the optional parameter `Set=<value>&` is also appended. It is not possible to preset all counters to individual counter states in one command.

Example for setting all counters to the value 12345 for a Web-IO with 12 inputs:

```
GET /counterclear?PW=blue&Set=12345&
```

in reply the Web-IO sends

```
counter0;12345;12345;12345;12345;12345;12345;
12345;12345;12345;12345;12345;12345 +(0x00)
```

At hoc reading of inputs, outputs and counters

By using the command `allout` you can query the complete process map of the Web-IO.

Example:

```
GET /allout?PW=blue&
```

in reply the Web-IO with 12 Inputs and 12 Outputs sends for example

```
input;0C3B;output0842counter;3974;453;99;0;0;984;
712;4;334;1076;0;6543 +(0x00)
```

Communication between Web-IO Analogue and Application

Querying the IO channels

Reading the IO channels is usually done in polling mode. This means: the client application requests the desired values with the help of the corresponding command string.

Either the status of a specific IO channel can be requested or a request is made for both IO channels.

If only one IO channel is to be queried, the number of the IO channel is added to the keyword single. For the query of all channels, indexing is waived. In contrast to the digital Web IOs, the indexing of the IO channels in the analogue Web IO does not start at 0 but at 1.

In response, the Web IO sends back the current or voltage value with three decimal places followed by a space and the unit. When both IO values are queried, they are returned separated by a semicolon.

Example of polling IO channel2 at 7.5 mA current flow :

```
GET  /single2&
```

The Web IO then sends back:

```
7,500 mA+(0x00)
```

Example of polling both IO channels at 15.340 mA and 7.5 mA current flow :

```
GET  /single&
```

The Web IO then sends back:

```
15,340 mA ;7,500 mA+(0x00)
```

As an alternative to the `single` command, the `output` command can also be used to read out the status of the IO channels (even if the channels are configured as inputs).

The `output` command is used in the same way as `single`. However, the Web IO prefixes the response with the word output and, if a channel is requested, with the corresponding index separated by a semicolon.

Example of polling IO channel2 at 7.5 mA current flow :

```
GET  /output2&
```

The Web IO then sends back:

```
output2;7,500 mA+(0x00)
```

Example of polling both IO channels at 15.340 mA and 7.5 mA current flow :

```
GET  /output&
```

The Web IO then sends back:

```
output;15,340 mA ;7,500 mA+(0x00)
```

Set current/voltage value of an output

As with read access, indexing is used. The number of the output to be set is appended to the keyword `outputaccess`.

In addition, the parameter `PW=` is used to pass the password and the parameter `State=` to pass the value. The parameter State is transferred with a maximum of three decimal places and without a unit.

The parameter set is separated from the command by `?` and each parameter ends with `&`.

Example:

```
GET /outputaccess1?PW=blau&State=15,340&
```

sets output 1 to 15.340 mA.

As a response, the Web IO sends e.g.

```
GET /outputaccess1?PW=blau&State=15,340& +(0x00)
```

Web-IO Climate, VOC and Application

Querying the IO channels

Reading the climate values is also usually done in polling mode. This means: the client application requests the desired values with the help of the corresponding command string.

Either the value of a specific channel can be requested or a request is made for all existing channels.

If only one channel is to be queried, the number of the channel is added to the keyword single. If all channels are to be queried, indexing is omitted. Unlike the digital Web IOs, the indexing of channels in the Web IO Climate does not start at 0 but at 1.

As a response, the Web IO sends back the current temperature, humidity, air pressure or air quality value with three decimal places followed by a space and the unit. If all values are requested, they are returned separated by a semicolon.

Example for the query of channel 2 (humidity) of a web thermo-hygrobarometer:

```
GET  /single2&
```

The Web Thermo-Hygrobarometer then sends back:

```
28,6% +(0x00)
```

Example for the query of all channels:

```
GET  /single&
```

The Web Thermo-Hygrobarometer then sends back:

```
23,6°C;28,5%;988,8hPa+(0x00)
```

Query and set date and time

Querying date and time

Use the command `time` to query the set date and time of the Web-IO.

The request

```
GET /time?PW=blue&
```

causes the Web-IO to reply with

```
01.08.2016,15:14:59
```

Setting the date and time

The system time of the Web-IO can be set using the command `settime`. The parameter time is used to send the desired settings formatted

```
time=DD.MM.YYYY,hh:mm:ss
```

Example:

```
GET /settime?PW= blue&time= 01.08.2016,15:14:59
```

the Web-IO replies with

```
01.08.2016,15:14:59
```

Reading the error memory

When the Web-IO detects one or more errors while processing one of its tasks, these are stored and can be viewed under Diagnostics in the navigation tree.

In some applications it is desirable to handle error management not manually by viewing a website, but rather to evaluate errors in automated fashion in a program.

For this the Web-IO provides a few command strings that can be used with either TCP or UDP.

Creating a list of possible error messages

The Web-IO can manage a maximum of 127 different error types. The actual number may vary up to this limit depending on the firmware version.

To receive an overview of the possible errors, you can send the command

```
GET /diaglistx?PW=password&
```

to the Web-IO.

In place of `x` a value between 0 and 127 is used. Instead of password enter the administrator or operator password.

The Web-IO replies to the request with the corresponding error message.

```
diaglistx;errortext
```

Example:

If the application sends (password = "`blue`")

```
GET /diaglist2?PW=blue&
```

the Web-IO replies with

```
diaglist2;Formatfehler in der DNS Anfrage
```

You can use the `diaglist` command with a *for next* loop to read out all possible error messages and store them as a static list. In binary mode for example you can only determine how many errors are currently present. In addition the associated error numbers are transmitted. Using the static error list a readable error can then be output.

Direct evaluation of the current errors
To get the current error status of the Web-IO you can use the command

```
GET /diagnosis?PW=password&
```

The Web-IO responds with

```
diagnosis;iiii;zzzzzzzz;yyyyyyyy;xxxxxxxx
```

In `iiii` the Web-IO returns the number of current errors in 4-place hex format.

`zzzzzzzz;yyyyyyyy; xxxxxxxx` may be ignored for normal, standard applications.

For expert programmers: `zzzzzzzz; yyyyyyyy; xxxxxxxx` *are three 32-bit values, each in 8-place hex format. The resulting 64 bits substitute for the 64 possible error types. By using the static error list (see GET /diaglist...) the individual bits can be decoded. The LSB appears to the right in the x range and the MSB to the left in the y range.*

To get the current errors as an error text, the Web-IO provides the command

```
GET /diagnosisx?PW=&
```

Here x is the index for the error in decimal format beginning with 1.

The reply from the Web-IO then looks as follows:

```
diagnosisx;errortext
```

In contrast to the error texts which are read from the Web-IO using `diaglist`, the error text for `diagnosis` may also contain dynamic elements such as IP addresses or port numbers.

Example:

The application sends:

```
GET /diagnosis?PW=blue&
```

The Web-IO responds:

```
diagnosis;0005;00400040;008A0000;00000000
```

The application sends:

```
GET /diagnosis1?PW=blue&
```

The Web-IO responds:

```
diagnosis1;Mail Server antwortet nicht.
```

The application sends:

```
GET /diagnosis2?PW=blue&
```

The Web-IO responds:

```
diagnosis2;Das Versenden der Mail wurde abgebrochen und wird wiederholt.
```

The application sends:

```
GET /diagnosis3?PW=blue&
```

The Web-IO responds:

```
diagnosis3;Ziel IP-Adresse unbekannt: 172.16.232.8.
```

The application sends:

```
GET /diagnosis4?PW=blue&
```

The Web-IO responds:

```
diagnosis4;Watchdog Timer abgelaufen!
```

The application sends:

```
GET /diagnosis5?PW=blue&
```

The Web-IO responds:

```
diagnosis5;TCP Client Alarm: Server nicht erreicht.
```

Clearing the error memory

The list of errors occurring during runtime remains stored in the Web-IO, even if the error is no longer present at the time of query.

To clear the error memory, use the command

```
GET /diagclear?PW=password&
```

The Web-IO responds with

```
diagnosis;0000;00000000;00000000;00000000
```

if there are no current errors.

TCP/IP Sockets - Binary

In addition to access using command strings, the Web-IO also provides four socket accesses for binary data exchange.

All four accesses work independently of each other. Under *Communication paths >> Socket-API* you can enable the binary accesses and specify which if the three following modes to use:

- TCP server
- TCP client
- UDP peer

For the various functions such as reading the inputs, setting the outputs etc. the Web-IO defines binary structures. Access is only by exchanging these structures.

TCP server

Under *Communication paths >> Socket-API* in the desired binary area select *BINARYx* TCP-Server as the Socket-Type and specify which local Port to use for accepting the connection.

Unlike TCP Server mode for command strings in which up to 8 clients can connect to the common server port, the Web-IO permits only one connection for the respective binary access. All further connection attempts are rejected as long as a client is connected.

Binary access can be password protected if desired.

When setting input triggers and there is an existing connection, the Web-IO automatically informs the application of changes to the selected inputs.

To switch the outputs via binary access as well, this must be checked under *Enable outputs for Binaryx-Sockets*.

TCP client

If *BINARYx TCP-Client* is selected as the Socket-Type, the Web-IO opens a connection to a TCP server application when there is a change to one of the inputs selected under Input/Trigger.

Under Server-IP enter the IP address or host name of the destination server and under *BINARYx Server Port* specify which TCP port the connection should be opened on.

In normal situations the *BINARYx local Port* should be configured for AUTO. This means the sender port increments dynamically at each connection, which is helpful in dealing with firewalls. Alternately you can also use a randomly selected, fixed sender port.

Under *Inactive Timeout* you can specify after how much time with no activity on the inputs a connection should be closed. A value of 0 means the connection remains open until the opposite endpoint closes it.

To switch the outputs via binary access as well, this must be checked under *Enable outputs for Binaryx-Sockets*.

UDP peer

If *BINARYx UDP-Peer* is selected as the Socket-Type, the Web-IO opens a connection to the UDP port set under *UDP local port* for receiving UDP datagrams.

All datagrams sent by the Web-IO go to the IP address entered under *Remote-Peer-IP* on the UDP port configured as U*DP remote port*.

When there is a change to the inputs defined as *Input-Trigger*, corresponding datagrams are sent to the configured remote peer.

To switch the outputs via binary access as well, this must be checked under *Enable outputs for Binaryx-Sockets*.

BINARY – The IO structures

Regardless of which BINARY Socket Type was selected, there are a clear number of binary structures (variable fields) for communication between the application and the Web-IO.

Such structures are provided for the following functions:

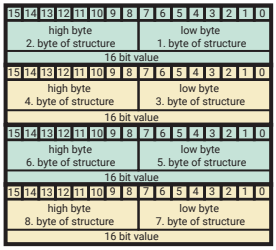
- Password prompt and sending
- Reading the inputs
- Reading the inputs and outputs
- Setting the outputs
- Reading the counters
- Resetting the counters
- Configuring cyclical and automatic messaging when there is a state change

The user program uses the convenient socket interface (Windows: WinSock, UNIX, Linux: Berkley Sockets) for exchanging the data in the form of these IO structures with the Web-IO and over the network via TCP/IP.

Definition of the structures

To be able to unambiguously identify and evaluate the content of a packet, in BINARY mode all the data must be sent to the Web-IO in the form of these IO structures.

All structures begin with the same header, which consists of the following 4 WORDS (16-bit_Integer):



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

Start_1, Start_2

Start_1 and *Start_2* are there for reasons of compatibility with earlier models, but they are not used. Both values are always 0.

StruktTyp

The value *struct_typ* identifies the structure. Both the PC application and the Web-IO use the value *struct_typ* when data are received to determine how the structure should be evaluated.

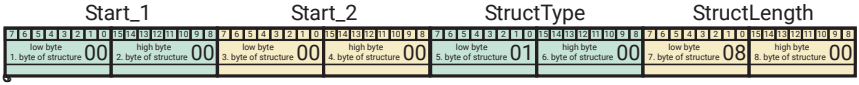
StructLength
length indicates the total length of the structure in bytes, i.e. including th first 4 WORDs.

Note: The following applies to all IO structures:

A word corresponds to 16bit integer.
A char corresponds to one byte (8bits)
A long corresponds to a 32bit integer

Hexadecimal format 0x in front of the value

For sending and receiving the variable Low-Byte first applies to all structure variables.



In the examples all numbers are in hex format!

Password protected access

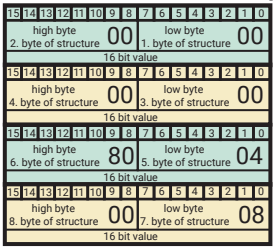
In TCP Server mode binary access can be protected from unauthorized attempts using a password protection.

If *Password protected access* is checked under *Communication paths >> Socket access > TCP/UDP Sockets BINARY-Mode* in the *BINARYx TCP-Server* area, a multi-step login procedure is required.

Unauthorized access attempt

If a client application attempts to begin communication with the Web-IO using the password protected binary access, the Web-IO sends the structure AuthRequired.

The structure Authrequired



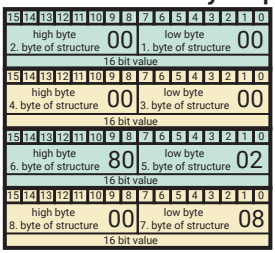
Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

When the *AuthRequired* structure is received, the client application must start with the login procedure.

Login procedure

To get access to the IOs with password protection, the client application must prompt a key. This is done by sending the *KeyRequest* structure.

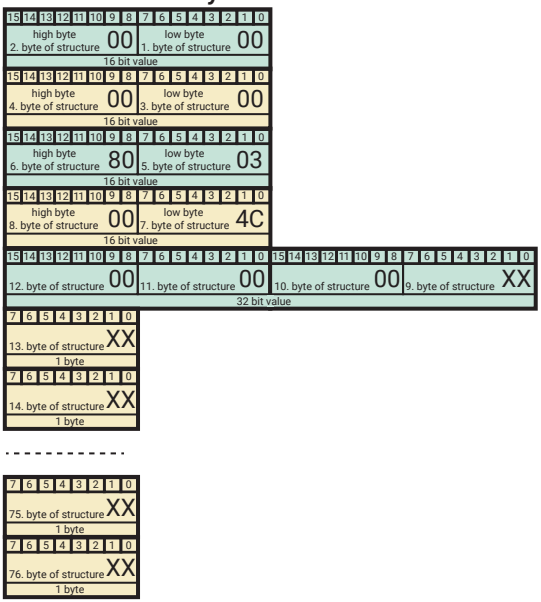
The structure KeyRequest



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

When the *KeyRequest* structure is received the Web-IO generates a maximum 64-byte long key to the Web-IO which is embedded in the *Key* structure.

The structure Key

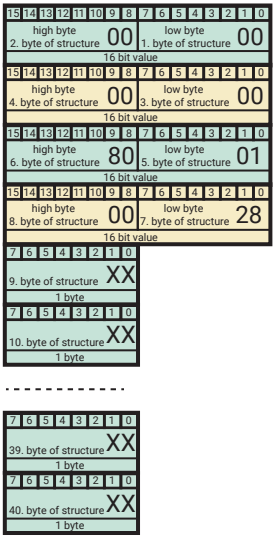


Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
KeyLength	Long Number of KeyBytes actually used
KeyBytes	Long byte array always 64 bytes, even if the actual key is shorter

How many bytes long the key actual is is indicated by the variable *KeyLength*.

The structure Login

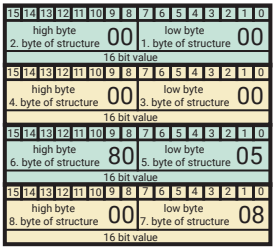
The client application must now generate an MD5 hash from the password assigned for the Web-IO and the received key.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
MD5-Hash	Long byte array 32 bytes MD5 hash of password + key

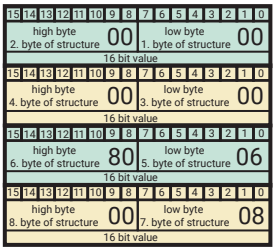
The MD5 hash sum is sent to the Web-IO using the Login structure. If the Web-IO accepts the login, it sends the *AuthOK* structure to the client application and normal data exchange can begin.

The structure AuthOK



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

The structure AuthFail



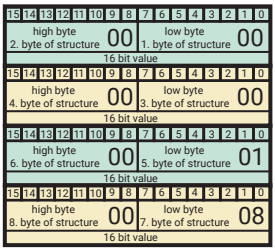
Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

If the login is incorrect the Web-IO sends the *AuthFail* structure and closes the TCP connection.

Digital IO-Access with Binary Structures

The structure ReadRegister

Sending this structure to the Web-IO causes it to send the status of Inputs 0 - 11 to the application program. The packet consists only of these four WORDs. This structure is used by the user program and the Web-IO always responds by sending the WriteRegister structure.

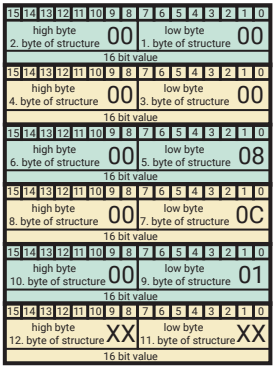


Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

The structure WriteRegister

This structure is used to send the state of the inputs or outputs for the Web-IO 12xDigital. If the application program sends this structure to the Web-IO, the Web-IO sets the outputs corresponding to the value transmitted in *value*.

If the Web-IO sends this structure to the user program, *value* has the value corresponding to the input state.

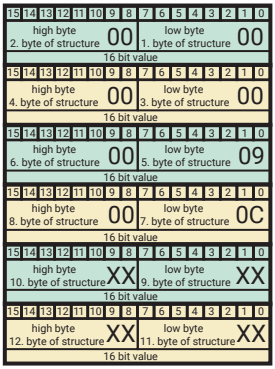


Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Amount	Word always = 1
Value	Word Binary equivalent of the input status

For Web IOs with relay outputs, a pause of at least 200ms must be observed between two switching processes at an output!

The structure SetBit

This structure enables the setting of individual outputs at the Web IO. If, for example, the entire process status is not mapped in the user programme, individual outputs can be set without changing the value of the others. The bits 0..11 in set_bits and value correspond to the corresponding outputs. This structure is only used by the user programme.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Mask	Word Bits that are to be changed = 1
Value	Word Binary equivalent of the output status

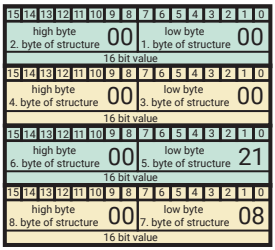
Beispiel.:

```
set_bits = 0x0124 / value = 0x0104
```

Output 2 and Output 8 (counting Output0..1) are set to ON and Output 5 is set to OFF. All other outputs are not changed.

The structure RegisterRequest

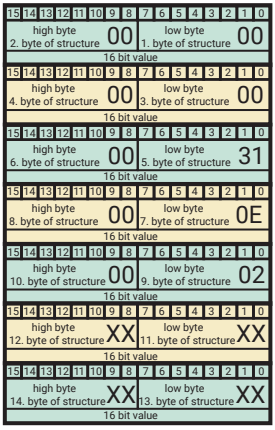
The user programme sends this structure to the Web IO in order to be able to read the contents of inputs and outputs in overview. The Web IO always responds with the RegisterStateert I/O structure.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

The structure RegisterState

The Web IO uses this structure to transmit the content of the inputs and outputs. This structure is only sent if the user application has sent the Register Request structure to the Web IO.

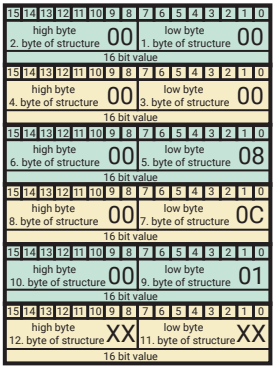


Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
DriverID	Word always = 2
InputValue	Word Binary equivalent of the input status
OutputValue	Word Binary equivalent of the Output status

The structure Send Mode

This structure is used to specify the trigger conditions the Web-IO 12xDigital uses to send the status of the inputs to the user program. There are basically three possibilities, but they may be combined with each other:

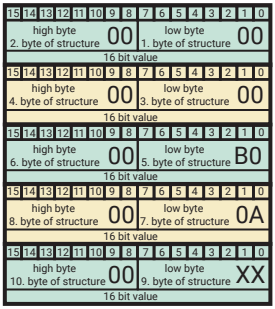
1. The user program polls the Web-IO by sending the READ structure
2. The Web-IO sends the WriteRegister - structure with the status of the inputs in a configurable interval
3. The Web-IO sends the WriteRegister - structure with the status of the inputs after a state change of the configured inputs



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Amount	Word always = 1
Value	Word Binary equivalent of the input status

The structure ReadCounter

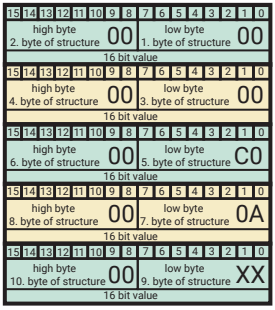
The user program sends this structure to the Web-IO to request the counter state of a certain input counter. Which input is intended is sent in the variable *counter_index*. The Web-IO always replies with the *Counter* structure.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Counterindex	Word Input number

The structure ReadClearCounter

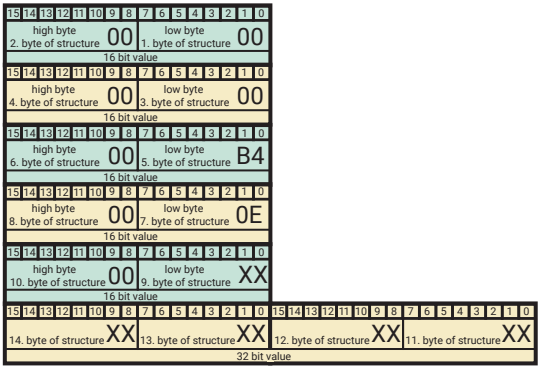
The application program sends this structure to the Web-IO in order to request the counter status of a particular input counter and then immediately set to counter to 0. Which input this involves is transmitted in the variable *counter_index*. The Web-IO always replies with the structure *Counter*.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Counterindex	Word Input number

The structure Counter

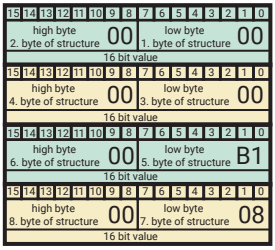
With this structure the Web-IO sends the counter state of the input counter specified in *counter_index*.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
CounterIndex	Word Nummer des Inputs
CounterValue	Long Zählerwert

The structure ReadAllCounter

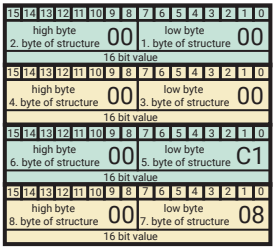
The user program sends this structure to the Web-IO to request the counter states of all the inputs in a data packet. The Web-IO always replies with the structure *All-Counter*.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

The Structure ReadClearAllCounter

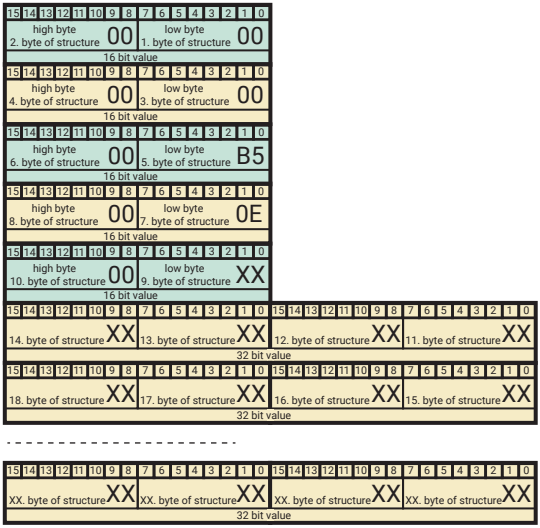
The application program sends this structure to the Web-IO in order to request the counter states of all inputs in a data packet and then immediately sets the counters to 0. The Web-IO always replies with the structure *AllCounter*.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

The structure AllCounter

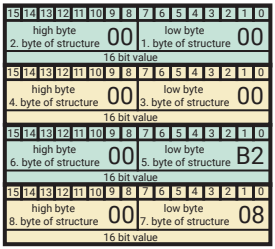
The Web-IO uses this structure to send the counter states of all the inputs at one time.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
CounterNoOf	Word Numner of Counters
CounterValue1	Long Counter value 1
CounterValue2	Long Counter value 2
CounterValue*n*	Long Counter value n

The structure ClearCounter

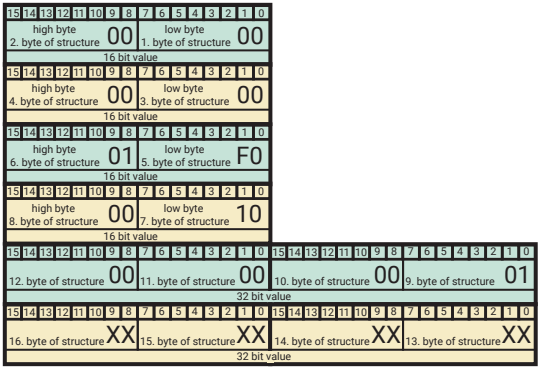
The user program sends this structure to the Web-IO to reset the counter state of a certain input counter to 0. Which input is intended is sent in the variable counter_index



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

The struktur Options

This structure is used to set certain options in the Web IO. For this purpose, 32 bits are available in the options variable.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Version	Long Currently 1
Options	Long Binary coding

In the present version of the Web IO, only bit 0 of the options variable is used.

- Bit 0 = 1 //the Web IO returns the RegisterState structure when an output is set.
- Bit 0 = 0 //the Web IO does not give any feedback when an output is set.

In order to reliably receive the status after setting the output(s) in the feedback, there should be a time of at least 50ms between two output-changing accesses.

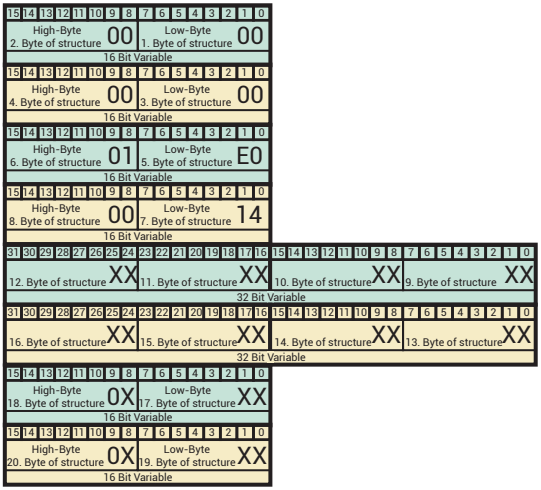
IO structures for reading the logger content (#57756)

The data records stored in the Web-IO 4.0 Digital Logger can also be read by using binary structures.

Even if you do not want to access the inputs or outputs, but only want to read out the logger, Binary1 must be activated in the Web-IO menu under *Communication paths >> Socket API >> Binary1*.

The struktur LOGGER_REQUEST

The *LOGGER_REQUEST* structure is used to initially transfer the time period for the desired data records to the Web-IO Digital Logger.

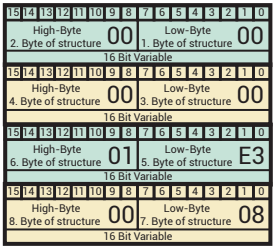


Start_1	Integer Always = 0
Start_2	Integer Always = 0
StructType	Integer Struktur type
StructLength	Integer Length in bytes
StartTime	Long (Seconds) from 01.01.1970
StopTime	Long (Seconds) from 01.01.1970
StartTime ms	Integer Milliseconds
StopTime ms	Integer Milliseconds

The start and stop times are specified in seconds from 01.01.1970 00:00:00. In addition, the ms (in steps of a hundred) up to the next full second can be specified for both times.

The struktur LOGGER_ NOACCESS

The content of the data logger can only be read by one instance at a time. If another read operation is already active, the Web-IO Digital Logger responds with the *LOGGER_NOACCESS* structure.

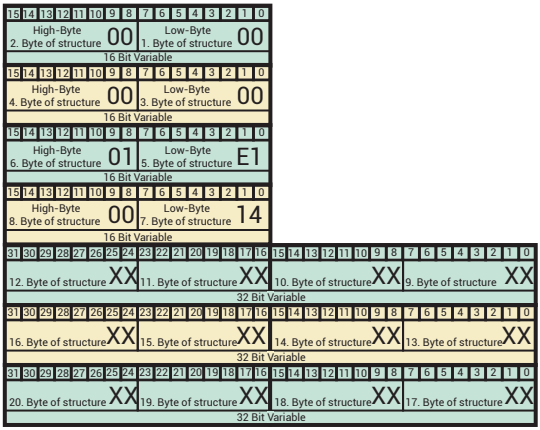


Start_1	Integer Always = 0
Start_2	Integer Always = 0
StructType	Integer Struktur type
StructLength	Integer Length in Bytes

In this case, the connection should be terminated and access to the logger should be attempted again at a later time.

The struktur LOGGER_CHECK

If access to the logger is not assigned, the structure *LOGGER_REQUEST* is answered by the Web-IO Digital Logger with the structure *LOGGER_CHECK*.



Start_1	Integer Always = 0
Start_2	Integer Always = 0
StructType	Integer Struktur
StructLength	Integer Length in Bytes
LoggerState	Long Check state
MemSize	Long Memory size
Memprogress	Long Search progress

The Web-IO Digital Logger must first check how many data records are stored for the desired time period. This can take some time depending on the memory utilization and the size of the time period. To inform the application about the current status of the data set search, the Web-IO Digital Logger returns the corresponding information with the *LOGGER_CHECK* structure.

- There are 4 states for LoggerState:

  0 = No search active (yet)

1 = Initialization of the search process
2 = Loading a memory area to be searched
3 = Recognizing the data records
4 = Evaluation completed

- In *MemSize* the Web-IO Digital Logger returns the total size of the memory to be scanned in bytes.
- The MemProgress parameter indicates how far the search has progressed.

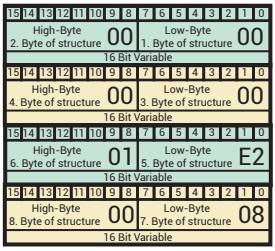
As long as the return value for logger_state is not equal to 4, the application must respond with the *LOGGER_CHECK* structure, whereby *LoggerState*, *MemSize* and *MemProgress* are set to 0.

LOGGER_CHECK structures are exchanged until LoggerState has the value 4.

The struktur LOGGER_DATAREQUEST

After the *LOGGER_CHECK* structure has been received with *LoggerState* = 4, the data records can be requested.

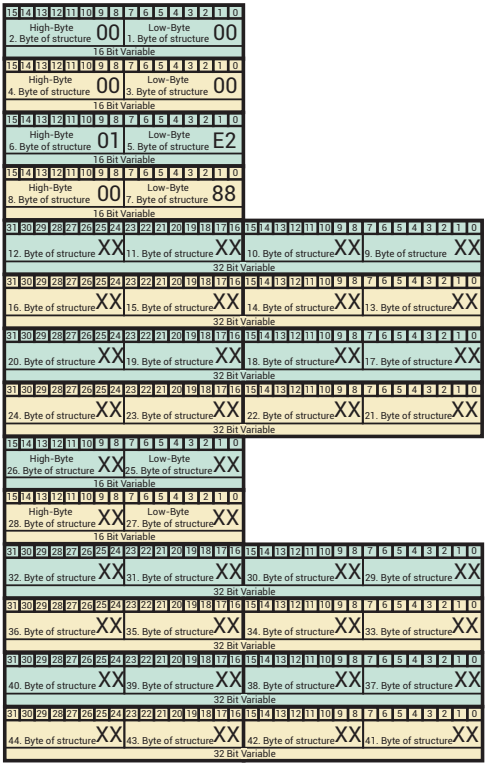
This is done by sending an *EADriver* structure with structure type 0x01E2 from the application to the Web-IO Digital Logger.



Start_1	Integer Always = 0
Start_2	Integer Always = 0
StructType	Integer Struktur type
StructLength	Integer Length in Bytes

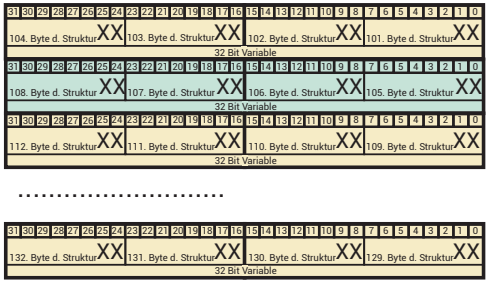
The struktur LOGGER_DATASET

The Web-IO Digital Logger starts sending the data sets after receiving the *LOGER_DATAREQUEST* structure. Each individual data set available for the desired time period is sent within a *LOGGER_DATASET* structure.



Start_1	Integer Always = 0
Start_2	Integer Always = 0
StructType	Integer Struktur type
StructLength	Integer Length in Bytes
MaxCount	Long Qty. of Datasets
ProcessCount	Long Current Datasets
TimeStamp	Long (Seconds) ab 01.01.1970
TimeStamp ms	Long + Milliseconds
Outputs	Integer Output bit pattern
Inputs	Integer Input bit pattern
Reserve	Long not used
Alarms1-30	Long Alarm bit pattern
Counter0	Long Counter value
Counter1	Long Counter value

.....



Counter14	Long Counter value
Counter15	Long Counter value
ErrorCounter0	Long Bidirectional Counting errors

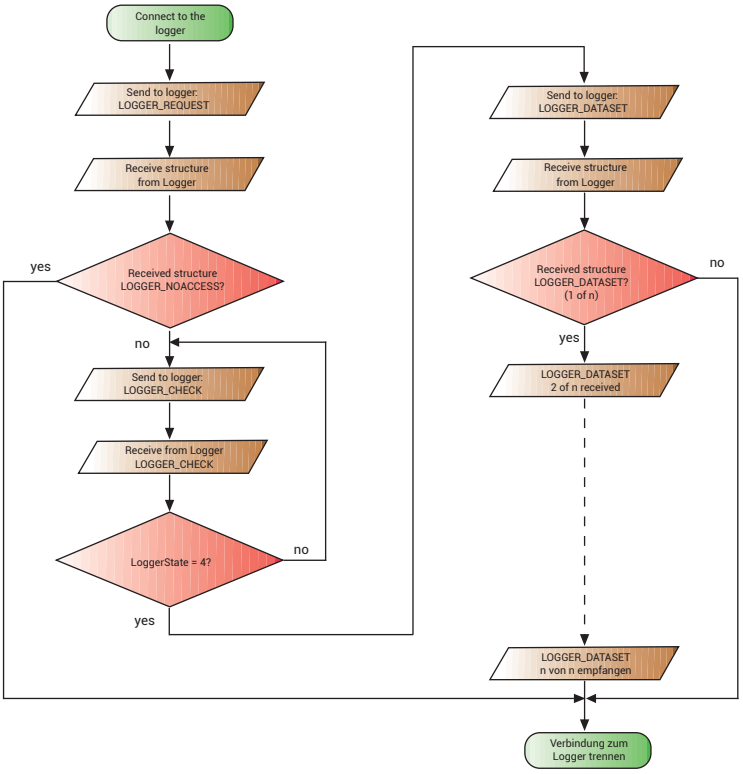
.....

ErrorCounter7	Long Bidirectional Counting errors

- The MaxCount parameter shows the total number of data records, while ProcessCount shows which data record is currently being transferred.
- Each data record contains the appropriate time stamp with TimeStamp (seconds from 01.01.1970 00:00:00) and TimeStamp ms (milliseconds).
- The binary status of the inputs and outputs is transferred in the Inputs and Outputs fields.

- The alarm status is signaled in the first corresponding bits of Alarms.
- The structure contains the 16-fold counter array for the counter readings.
- In the event that the bidirectional counter mode is used and direction detection was not clearly possible, the counting errors are incremented in the 8-fold ErrorCounter array.
- The Web-IO Digital Logger sends all data records for the selected time period without a new request from the application side. This process cannot be stopped.

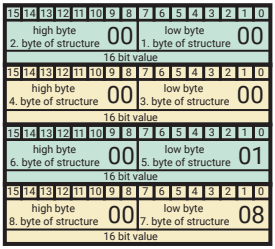
The exact procedure for reading out the logger is clearly illustrated in the following flow chart.



Analog I/O access with binary structures

The structure ReadRegister

Sending this structure to the Web-IO causes it to send the status of the ports to the user program.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

The package consists only of these four WORDs. This structure is only used by the user program and the Web-IO always reacts by sending the structure AnalogRegisterState.

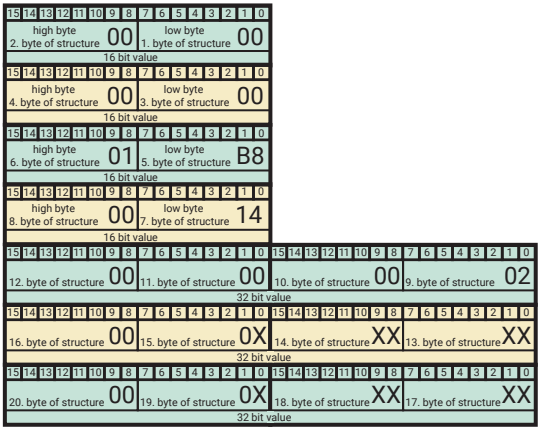
The IO structure AnalogRegisterState

The Web-IO Analog-In/Out transmits the status of both ports with this structure. This structure is sent if the user program has sent the ReadRegister structure to the Web-IO, or if an output value has been set with this structure.

This structure also serves to transmit the output values of the ports to be set. If the user program sends this structure to the Web-IO, the Web-IO sets the outputs according to the value transferred in Port 1 and Port 2.

Here the value is not transmitted in the configured unit, but always in 1/100,000 of the maximum value. An output value of 15.4mA must be transmitted as 77000 or 0x012CC8 .

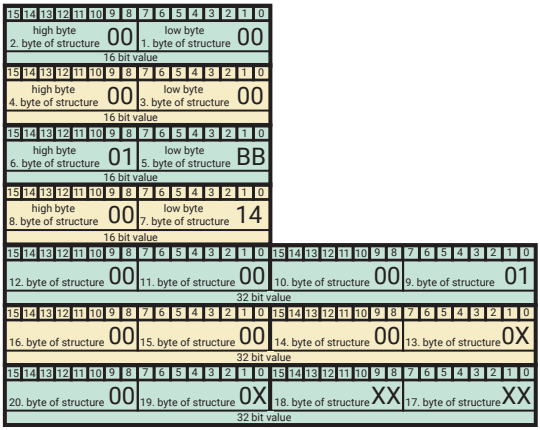
If the Web-IO sends this structure to the user program, port 1 and port 2 have the value corresponding to the input status.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Amount	Long always 2
Port1Value	Long Value in 1/100.000
Port2Value	Long Value in 1/100.000

The structure AnalogSingleRegister

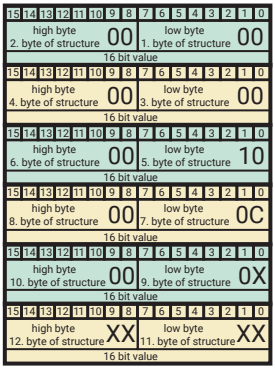
This structure is used to transmit the output value of a single port at Web-IO Analog-In/Out (Port 1 = 0, Port 2 = 1). The procedure is the same as for AnalogRegisterState.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Amount	Long always 1
Channel	Long Chanel no. (0 / 1)
PortValue	Long Value in 1/100.000

The IO structure Send Mode

This structure defines the trigger conditions with which the Web-IO Analog-In/Out sends the status of the ports to the user program. The trigger can be configured for status changes of both ports. The respective hysteresis for the trigger must be set in the web configuration.



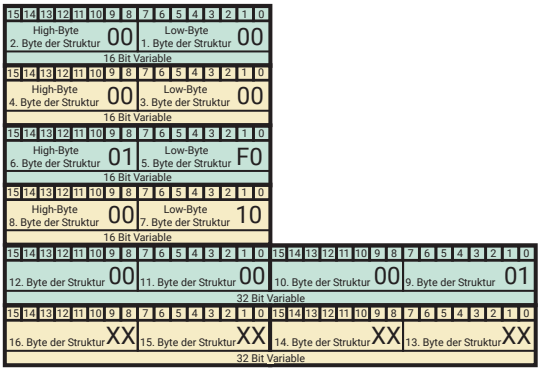
Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Mask	Word Bits for input trigger = 1
Interval	Word Interval in 100ms for Transmission of the

The following combinations can be configured as variable masks:

	Port 1	Port 2
0x0000	off	off
0x0001	on	off
0x0002	off	on
0x0003	on	on

The IO structure Options

This structure is used to set certain options in the Web-IO. For this purpose 32 bits are available in the variable options.



Start_1	Integer Immer = 0
Start_2	Integer Immer = 0
StructType	Integer Strukturtype
StructLength	Integer Länge in Bytes
Version	Long Version z.Zt. 1
Options	Long Binäre Kodierung

In der vorliegenden Version des Web-IO wird nur das Bit 0 der Variablen *options* verwendet.

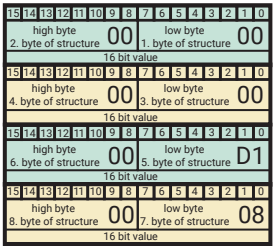
- Bit 0 = 1 //das Web-IO sendet die AnalogRegisterState Struktur zurück, wenn ein Output gesetzt wird.
- Bit 0 = 0 //das Web-IO gibt bei Setzen eines Output keine Rückmeldung.

Um in der Rückmeldung zuverlässig den Zustand nach Setzen des/der Output(s) zu bekommen, sollte zwischen zwei Output-verändernden Zugriffen min. eine Zeit 50ms liegen.

Device status via binary structures

The structure ReadDiagnosis

If the Web-IO determines that there is a communications or system error, the latter is listed on the HTML page diag and can be read using the browser. Since error management via browser for program-controlled applications is not always available, the error status of the Web-IO can be queried using the structure *ReadDiagnosis*

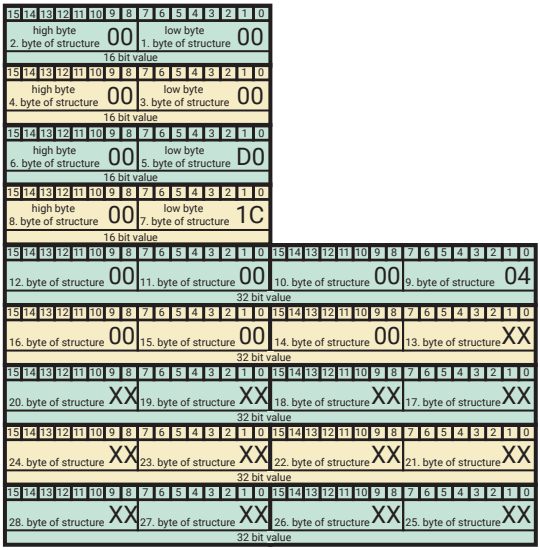


Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

The Web-IO replies with a Diagnosis type structure.

The structure Diagnosis

The Web-IO replies to the *ReadDiagnosis* structure with a Diagnosis type structure.



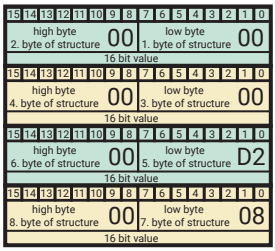
Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Version	Long currently = 4
ErrorCount	Long current error number
Errorbits1	Long binary error coding
Errorbits2	Long binary error coding
Errorbits3	Long binary error coding

The variable diag_error_count returns how many various errors are currently in the error memory. The Web-IO distinguishes a total of up to 64 different error sates, whereby each set bit of the two variables diag_errrorbits0 and diag_errorbits1 stands for an error type.

The exact text description can be obtained on TCP Port 80. For a more detailed description, see the section on socket programming with command strings.

The structure ClearDiagnosis

This structure clears the error memory of the Web-IO.

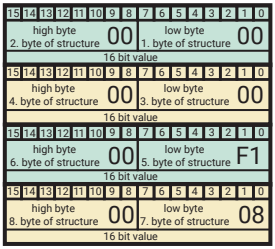


Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

Device features via binary structure

The structure InventoryRequest

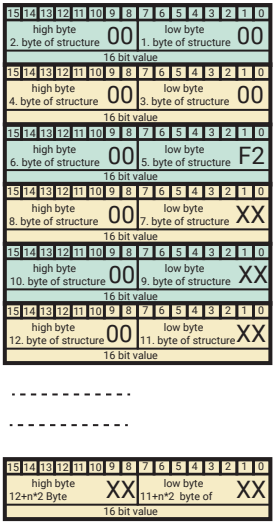
To query the IO equipment of the Web IO, the InventoryRequest structure is sent to the Web IO.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes

the structure Inventory

If the Web IO receives the InventoryRequest structure, it responds with the Inventory structure.



Start_1	Word always = 0
Start_2	Word always = 0
StructType	Word Structure type
StructLength	Word Length in bytes
Amount	Word Number of IO mode entries
IO-Mode	Word high byte = IO chanel no. low byte = IO chanel mode

. . . . . .

. . . . . .

In the Amount variable, the Web IO specifies how many IO mode entries the structure contains.

Each entry consists of the channel number as a high byte and the available IO mode as a low byte.

If more than one value or state can be retrieved from an IO channel, there is an entry for each retrievable parameter (e.g. for digital input, one entry for the input state and another for the counter value).

IO chanel mode
The IO modes are numbered as follows:

1 analogue input (4..20mA or 0..10V)
2 digital input
3 digital output (read and write access)
4 counter
5 digital output (read access only)
6 analogue output

IO chanel no
The IO channel number corresponds to the indexing of the individual inputs or outputs. Depending on the IO type, the indexing can start at 0 (e.g. for digital Web IOs) or 1 (e.g. for analogue Web IOs)

HTTP-Request

In addition to the classic socket access, the Web-IO can also be addressed directly via the HTTP access by HTTP requests

This access is disabled at in factory defaults and must be activated in the menu tree at Communication paths >> Web-API.

In addition, the Web-IO offers the option of uploading and saving your own, self-programmed website under visualizations >> My website. Your own website can be accessed via the menu tree under My Website or the name you have given. But there is also a direct call via the URL possible:

```
http: // <IP address of the Web-IO> / user
```

Supported requests (commands)

The following list shows all supported HTTP requests

Further details on access using web techniques such as AJAX or PHP can be found on the following pages.

Web-IO Digital

Kommandos	Parameter	Beschreibung
`/input`**`x`**	`?PW=`**`password`**`&` The administrator or user password must be used instead of `password`. If no password has been assigned, `PW=&` is entered. (applies to all commands!)	Input status request `x` optionally specifies the no. of the input and can be between 0 and 16 depending on the model. The feedback of the Web IO is a string starting with `inputx`; followed by the input status: `ON` = signal at the input and `OFF` = no signal at the input If `x` is omitted completely, the Web IO returns a bit pattern corresponding to the input signals in hexadecimal notation.

Kommandos	Parameter	Beschreibung
/counter**x**	?PW=**password**&	Counter value request x can be a value between 0-11 and indicates the input. The feedback from the Web IO is a string beginning with counterx;. The counter value of the selected counter is appended in decimal notation.
/counter	?PW=**password**&	Requesting all counter values The feedback of the Web IO is a string beginning with counter;. The counter values are added in decimal notation separated by semicolons.
/outputx	?PW=**password**&	Output status request x optionally specifies the No. of the output and can be between 0 and 16 depending on the model. The feedback from the Web IO is a string starting with outputx; followed by the output status: ON = signal at the output and OFF = no signal at the output If x is omitted completely, the Web IO returns a bit pattern corresponding to the output signals in hexadecimal notation.
/outputaccessx	?PW=**password**& [Mask=**XXXX**&] State=**ON**/**OFF**/**YYYY**& [NA=ON&] ON: Output = 1, OFF: Output = 0, TOGGLE: Change of state **XXXX**: Hex value between 0000 and 0FFF according to the bits that are to be set **YYYY**: Hex value between 0000 und 0FFF according to the output bit pattern.	Set one or more outputs x can be a value between 0-11 and specifies the output to be set. The feedback of the Web IO is a string starting with „output;“ followed by a bit pattern corresponding to the output signals in hexadecimal notation. The specification of Mask is optional. If Mask is not sent, the outputaccess command applies to all outputs. With NA=ON it is optionally achieved that no response to the outputaccess command is sent by the Web IO.
/counterclear**x**	?PW=**password**& [Set=**value**&] **value**: Counter preset, value between 0 and 2 billion	Sets the counter value of a counter. If the Set parameter is not sent, the default is 0. x can be a value between 0-11 and specifies the input whose counter is to be reset. The response of the Web IO is a string beginning with counterx;. The new count of the selected counter is appended in decimal notation. If x is not specified, all counters are set.

Kommandos	Parameter	Beschreibung
/allout	?PW=**password**&	Collective request of the input, output states and all counter values. The Web IO responds with a string of the following structure: input;0xxx;output;0xxx- ;counter;n0;n1;n2,... 0xxx corresponds to the status of the inputs or outputs in hexadecimal notation. n0, n1 etc. contain the counter states in decimal notation.
/time	?PW=**password**&	Returns the system time of the Web IO. Format: **DD.MM.YYYY,hh:mm:ss** **D**=Day, **M**=Month, **Y**=Year, **h**=Hour, **m**=Minute, **s**=Second
/settime	?PW=**password**& time=**DD.MM.YYYY,** **hh:mm:ss**&	Sets the system time of the Web IO to the value passed with time.
/diagnosis	?PW=**password**&	Requests the status of the diagnostic memory. The Web IO returns: diagnosis;0000;00000000; 00000000;00000000 the four-digit value indicates the number of stored messages. With the three eight-digit hexadecimal values, each set bit represents one of the 92 possible messages.
/diagnosis**x**	?PW=**password**&	With x, the index for one of the currently stored messages is specified. The Web IO sends the corresponding message text as a return. x must not be greater than the number of the currently present messages.
/diaglist**x**	?PW=**password**&	Returns the messages for the individual message bits (max. 92)
/diagclear	?PW=**password**&	Clears message memory
/errorclear	?PW=**password**&	Clears load errors and releases the affected outputs.

Web-IO Analog

Kommandos	Parameter	Beschreibung
/single**x**	keine	Request for the current current or voltage values in mA or V. x optionally indicates the no. of the IO channel and can be 1 or 2. The feedback of the Web-IO is a string that shows the value with three decimal places and unit [N]N.NNN mA or [N] N.NNN V If x is omitted completely, the Web IO returns the values of both channels separated by semicolons.
/output**x**	keine	gives the same result as GET /single**x** (even if IO channels work as input)
/outputaccessx	?PW=**password**& State=**N,NNN**& **N,NNN:** Strom- bzw. Spannungswert der am entsprechenden output eingestellt werden soll	Setting an output x can be 1 or 2 and indicates the output to be set. The feedback of the Web IO is a string in the format [N]N.NNN mA or [N] N.NNN V. and indicates the current value. Please note that the Web IO needs a few ms to set the desired value. Therefore, the value does not correspond to the desired value.
/time	?PW=**password**&	Returns the system time of the Web IO. Format: **DD.MM.YYYY,hh:mm:ss** **D**=Day, **M**=Month, **Y**=Year, **h**=Hour, **m**=Minute, **s**=Second
/settime	?PW=**password**& time=**DD.MM.YYYY,** **hh:mm:ss**&	Sets the system time of the Web IO to the value passed with time.
/diagnosis	?PW=**password**&	Requests the status of the diagnostic memory. The Web IO returns: diagnosis;0000;00000000; 00000000;00000000 the four-digit value indicates the number of stored messages. With the three eight-digit hexadecimal values, each set bit represents one of the 92 possible messages.
/diagnosis**x**	?PW=**password**&	With x, the index for one of the currently stored messages is specified. The Web IO sends the corresponding message text as a return. x must not be greater than the number of the currently present messages.
/diaglist**x**	?PW=**password**&	Returns the messages for the individual message bits (max. 92)
/diagclear	?PW=**password**&	Clears message memory

Web-IO Klima (Web-Thermometer, ...) und VOC

Kommandos	Parameter	Beschreibung
/single**x**	keine	Request for the current climate data x optionally indicates the no. of the sensor. The feedback of the Web-thermometer is a string that shows the value with one decimal digit and unit 24,0°C If x is omitted completely, the Web IO returns the values of both channels separated by semicolons.
/time	?PW=**password**&	Returns the system time of the Web IO. Format: **DD.MM.YYYY,hh:mm:ss** **D**=Day, **M**=Month, **Y**=Year, **h**=Hour, **m**=Minute, **s**=Second
/settime	?PW=**password**& time=**DD.MM.YYYY,** **hh:mm:ss**&	Sets the system time of the Web IO to the value passed with time.
/diagnosis	?PW=**password**&	Requests the status of the diagnostic memory. The Web IO returns: diagnosis;0000;00000000; 00000000;00000000 the four-digit value indicates the number of stored messages. With the three eight-digit hexadecimal values, each set bit represents one of the 92 possible messages.
/diagnosis**x**	?PW=**password**&	With x, the index for one of the currently stored messages is specified. The Web IO sends the corresponding message text as a return. x must not be greater than the number of the currently present messages.
/diaglist**x**	?PW=**password**&	Returns the messages for the individual message bits (max. 92)
/diagclear	?PW=**password**&	Clears message memory

An example for use of HTTP requests

The following example shows how you can construct a self-refreshing Web page for the Web-IO using JavaScript and

HTML - structure of a static web page

HTTP requests (AJAX). First the purely HTML section of the Web page, which serves in essence as a display foundation for AJAX:

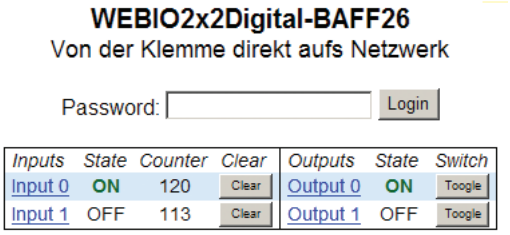
```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html;
charset=windows-1250">
  <title>Web-IO Digital, User</title>
  <style type="text/css">
    * { font-family:arial; }
    table { font-size:14px; }
    .borderLeft { border-left:1px solid #000000; }
    .button { font-size:9px; width:40px; }
    .ce { text-align:center; }
    .description { font-size:18px; text-align:center; }
    .info { font-size:10px; text-align:center; }
    .italic { font-style:italic; }
    .name { font-size:20px; font-weight:bold; text-align:center }
    .sideSpace { padding-left:5px; padding-right:5px; }
    .table { background-color:#d6e8ff; border-collapse:collapse;
          border:1px solid #000000; }
    .whiteBack { background-color:#ffffff; }
  </style>
  <script language="JavaScript" type="text/javascript">
```

this is actually the JavaScript section of the page, which is described in greater detail below.

```
  </script>
  </head>
<body onload="CommandLoop();">
<div class="name"><w&t_tags=device_name></div>
<div class="description"><w&t_tags=device_text></div>
<br>
  <form>
    <div align="center">
              <span id="pt"">Password: </span>
        <input id="pw" type="password" name="tfPassword" maxlength="31"
size="20"">
              <input  id="lb" type="button" value="Login"
onclick="setPassword()">
        </div>
  </form>
<table align="center" class="table">
  <tr class="italic whiteBack">
    <td class="sideSpace">Inputs</td>
```

```
    <td class="sideSpace">State</td>
    <td class="ce sideSpace20">Counter</td>
    <td class="ce">Clear</td>
    <td class="borderLeft sideSpace">Outputs</td>
    <td class="sideSpace">State</td>
    <td class="ce">Switch</td>
  </tr>
  <tr>
    <td class="sideSpace"><w&t_tags=input0></td>
    <td class="ce" id="input0">-</td>
    <td class="ce" id="counter0">-</td>
    <td class="ce sidespace">
      <input class="button" onclick="clearCounter(0);" type="button"
value="Clear">
    </td>
    <td class="borderLeft sideSpace"><w&t_tags=output0></td>
    <td class="ce" id="output0">-</td>
    <td class="sideSpace">
      <input class="button" onclick="setOutput(0);" type="button"
value="Toogle">
    </td>
  </tr>
  <tr class="whiteBack">
    <td class="sideSpace"><w&t_tags=input1></td>
    <td class="ce" id="input1">-</td>
    <td class="ce" id="counter1">-</td>
    <td class="ce sidespace">
      <input class="button" onclick="clearCounter(1);" type="button"
value="Clear">
    </td>
    <td class="borderLeft sideSpace"><w&t_tags=outpu1></td>
    <td class="ce" id="output1">-</td>
    <td class="sideSpace">
      <input class="button" onclick="setOutput(1);" type="button"
value="Toogle">
    </td>
  </tr>
</table>
</body>
</html>
```

Here is how this looks in the browser:



JavaScript and AJAX - Change content dynamically

The functions shown below are based on two basic techniques:

- Identification and later changing of an HTML object using a unique ID
- Communication with the server (Web-IO) after loading the Web page using HTTP requests

The JavaScript which is part of the Web page uses both these techniques.

First `maxi` and `maxo` are used to determine how many inputs and outputs are to be supported. A variable `applicationstep` is defined for the various program steps. 500ms is specified as the request interval, and a variable is entered for the password.

```
var maxi = 2;
var maxo = 2;
var applicationstep = 0;
var interval = 500;
var iopassword = ,';
```

The `HextoInt` function calculates hexadecimal strings to decimal values.

```
function HexToInt(HexStr)
{ var TempVal;
  var HexVal=0;
  for( var i=0; i<HexStr.length;i++)
  { if (HexStr.charCodeAt(i) > 57)
    { TempVal = HexStr.charCodeAt(i) - 55;
    }
    else
    { TempVal = HexStr.charCodeAt(i) - 48;
    }
    HexVal=HexVal+TempVal*Math.pow(16, HexStr.length-i-1);
  }
  return HexVal;
}
```

The function `CommandLoop` combines the HTTP requests and sends them to the `DataRequest` function.

```
function CommandLoop()
{ var commandstring ='';
  applicationstep++;
  switch(applicationstep)
  { case 1:
        commandstring = 'input?PW=' + iopassword + '&';
    break;
    case 2:
      commandstring = 'output?PW=' + iopassword + '&';
```

```
      break;
      case 3:
        commandstring = 'counter?PW=' + iopassword + '&';
        applicationstep = 0;
      break;
    }
    DataRequest(commandstring);
    maintimer = setTimeout("CommandLoop()", interval);
}
```

The `DataRequest` function is the heart of this JavaScript. It receives the HTTP request and sends it to the server (Web-IO). The `DataRequest` function also receives the reply from the Web-IO and sends it to the `updateDisplay` function.

```
  function DataRequest(SendString)
  { var xmlHttp;
    if( window.ActiveXObject )       // Internet Explorer
    { xmlHttp = new ActiveXObject( Microsoft.XMLHTTP" );
    }
    else if(window.XMLHttpRequest )  // Mozilla, Opera und Safari
    { xmlHttp = new XMLHttpRequest();
    }
    if (xmlHttp)
    { xmlHttp.onreadystatechange = function()
      { if (xmlHttp.readyState == 4)
        { if (xmlHttp.status == 200)
          { if (xmlHttp.responseText.length > 0)
            { updateDisplay(xmlHttp.responseText);
            }
            xmlHttp=null;
          }
        }
      }
      xmlHttp.open("GET", SendString, true);
      xmlHttp.setRequestHeader("Connection", "close");
      xmlHttp.setRequestHeader("If-Modified-Since", "Thu, 1 Jan 1970
00:00:00 GMT");
      xmlHttp.send(null);
    }
  }
```

The `updateDisplay` function evaluates the reply from the Web-IO and correspondingly adjusts the browser display. A check is made as to whether the Web-IO reply refers to inputs, outputs or counters. JavaScript uses `document.getElementById(ID)` to identify the objects for changing and adjusts their properties to the actual IO status

```
function updateDisplay(ReceiveStr)
{ var HexVal;
  var state;
  var ReceiveData = ReceiveStr.split(';')
  // Display Intput state
  if (ReceiveData[ReceiveData.length - 2].substring(0, 1) == 'i')
  { HexVal = HexToInt(ReceiveData[ReceiveData.length - 1]);
```

```
    for (var i = 0; i < 2; i++)
    { state = false;
      if ((HexVal & Math.pow(2, i)) == Math.pow(2, i))
      { state = true;
      }
      document.getElementById('input'+i).firstChild.data =( !state ) ?
'OFF':'ON';
      document.getElementById('input'+i).style.color
                                = ( !state ) ? '#000000' : '#006600';
      document.getElementById('input'+i).style.fontWeight
                                  = ( !state ) ? 'normal' : 'bold';
    }
  }
  // Display Output state
  if (ReceiveData[ReceiveData.length - 2].substring(0, 1) == 'o')
  { HexVal = HexToInt(ReceiveData[ReceiveData.length - 1]);
    for (var i = 0; i < 2; i++)
    { state = false;
      if ((HexVal & Math.pow(2, i)) == Math.pow(2, i))
      { state = true;
      }
      document.getElementById('output'+i).firstChild.data = ( !state ) ?
'OFF' : 'ON';
      document.getElementById('output'+i).style.color
                                = ( !state ) ? '#000000' : '#006600';
      document.getElementById('output'+i).style.fontWeight
                                  = ( !state ) ? 'normal' : 'bold';
    }
  }
  //Display Counter
  if (ReceiveData.length - maxi - 1 >= 0)
  { if (ReceiveData[ReceiveData.length - maxi - 1].substring(0, 1) == 'c')
    { for (var i = 0; i < maxi; i++)
      { document.getElementById('counter' + i).innerHTML
                          = ReceiveData[ReceiveData.length - maxi + i]
      }
    }
  }
  //Display cleared Counter
  if (ReceiveData[ReceiveData.length - 2].substring(0, 1) == 'c')
  { document.getElementById('counter'
  + ReceiveData[ReceiveData.length - 2].substring(7,
ReceiveData[ReceiveData.length -       2].length)).innerHTML =
ReceiveData[ReceiveData.length - 1];
  }
}
```

The `setOutput` function sends the corresponding command for toggling the selected output to the `DataRequest` function.

```
function setOutput(iNr)
{ var commandstring =
'outputaccess'+iNr+'?PW='+iopassword+'&State=TOGGLE&';
  DataRequest(commandstring);
}
```

The `clearCounter` function sends the corresponding command for clearing the

counter state of the selected counter to the `DataRequest` function.

```
function clearCounter(iNr)
{  DataRequest('counterclear'+iNr+'?PW='+iopassword+'&');
}
```

The `setPassword` takes the entered password and writes it to the `iopassword` variable, which is a component of the command string.

```
function setPassword()
{  iopassword = document.getElementById('pw').value;
   document.getElementById('pw').value = '';
}
```

The AJAX technique shown here, which is based on HTTP requests, can only be used if the web page is can only be used if the web page is loaded directly from the Web IO. For security reasons the most common browsers prevent HTTP requests that are sent subsequently from being to other servers than the one from which the original web page was loaded was loaded. This technique is called Same Origin Policy (SOP).

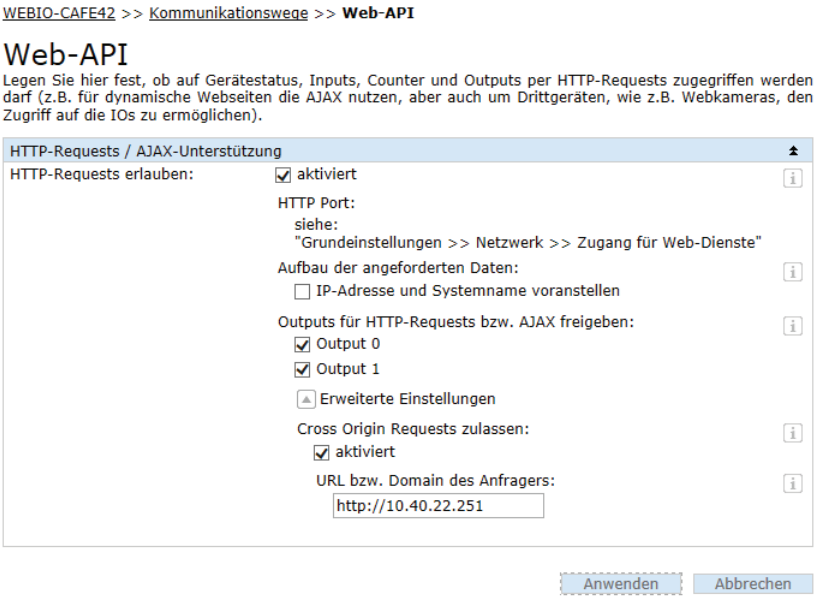
A decisive disadvantage of this SOP restriction is that the normal AJAX technology does not allow a common web page to display, for example, the states of two different web IOs.

CORS - Cross Origin Resource Sharing

To circumvent the SOP restriction, cross-origin resource sharing is used. sharing. In order to be able to use this technique with the Web IO, Cross Origin must be activated under Communication Routes >> Web API in the Advanced Settings area, Cross Origin must be activated.

The URL of the server from which the original web page was loaded must be entered as the URL of the requestor, from which the original web page was loaded.

If the web page is loaded directly from the hard disk of the local PC, „*" must be entered as the URL of the requester. „*" is synonymous with all URLs allowed.



In the JavaScript part, the DataRequest function in particular must be structured differently.

```
function DataRequest(device, SendString)
{ var cor;
  if(window.XDomainRequest)
  {  cor = new XDomainRequest();
  }
  else
  { cor = new XMLHttpRequest();
  }
  if (cor)
  { cor.onreadystatechange = function()
    { if(cor.readyState == 4)
      {  updateDisplay(cor.responseText);
      }
    }
    cor.open('GET', 'http://'+device+'/'+SendString);
    cor.send();
  }
  else
  { alert('Your Browser does not support Cross Origin Request');
  }
}
```

In contrast to the original function, the extended request method `XDomainRe-`

quest() or XMLHttpRequest() is used. When formulating the request, the complete URL including the IP address or the host name must be passed (variable device).var device = '10.40.22.242';

The functions that call DataRequest must of course be adapted accordingly.

```
function CommandLoop()
{  var commandstring ='';
   applicationstep++;
   switch(applicationstep)
   {  case 1:
         commandstring = 'input?PW=' + iopassword + '&';
      break;
      case 2:
        commandstring = 'output?PW=' + iopassword + '&';
      break;
      case 3:
        commandstring = 'counter?PW=' + iopassword + '&';
        applicationstep = 0;
      break;
   }
   DataRequest(device, commandstring);
   maintimer = setTimeout("CommandLoop()", interval);
}

function setOutput(iNr)
{  var commandstring=outputaccess'+iNr+'?PW='+iopassword+'&State=TOGGLE&';
   DataRequest(device, commandstring);
}

function clearCounter(iNr)
{  DataRequest(device, 'counterclear'+iNr+'?PW='+iopassword+'&');
```

Using W&T Tags

As described in section „5.1.4 Labeling and texts“ the Web-IO itself can be used to freely name and label inputs, outputs and counters.

A corresponding display in the browser is accomplished by using W&T tags. W&T tags are placeholders which are replaced by the Web-IO when sending the Web page to the browser using the stored names. This makes it possible for one and the same Web page to have a different appearance in different Web-IOs.

These tags consist of <w&t_tags= and the actual function invocation.

```
<w&t_tags=time>
```

for example shows the current system time and the date in the browser.

There is a special feature of the W&T tags which show the status of the outputs and the counter states of the input counters.

`<w&t_tags=ox>` and `<w&t_tags=cx>`

When there is an administrator or operator login the displayed contents (ON/OFF or counter state) has a hyperlink. Clicking on this link changes the state of the outputs or sets the counter to 0.

So that the changed state is then displayed, the browser automatically reloads the Web page.

The following W&T tags are available:

W&T-Tag	Beschreibung/Funktion
`<w&t_tags=device_name>`	Inserts the name assigned for the Web-IO in the web site.
`<w&t_tags=device_text>`	Inserts the description defined for the Web-IO in the web site.
`<w&t_tags=location>`	Inserts the location assigned for the Web-IO in the web site.
`<w&t_tags=contact>`	Inserts the contact assigned for the Web-IO in the web site.
`<w&t_tags=inputx>`	Inserts the name specified for input no. x. x can be a number between 0-11 and indicates which input the invoke refers to. This applies as well to the output and counter invokes described below.
`<w&t_tags=ix>`	Indicates the state (ON/OFF) of the input corresponding to x.
`<w&t_tags=cx>`	Fügt den Zählerstand des Counters für Input *x* in die Webseite ein.
`<w&t_tags=outputx>`	Inserts the name specified for Output x

W&T-Tag	Beschreibung/Funktion
<w&t_tags=ox>	Shows the state (ON/OFF) of the output corresponding to x. When logging in with Operator or Administrator rights, the state indication is given a hyperlink. Clicking on this link changes the state of the corresponding output and refreshes the web site
<w&t_tags=time>	Inserts the system time and data of the Web-IO in the Web site.

HTTP requests outside the browser

The HTTP requests shown here can also be used with other web techniques such as PHP.

The web IOs can also be controlled via HTTP requests using tools such as cURL.

When calling cURL, the URL should be placed in inverted commas when passing the parameters:

Example:

```
curl "http://10.40.22.242/outputaccess1?PW=&State=TOGGLE&"
```

A combination of cURL and PHP is also possible:

```
<?php
  $URL = 'http://10.40.22.242/outputaccess1?PW=&State=TOGGLE&';
  $ch = curl_init();
  $options = array(
  CURLOPT_URL            => $URL,
  CURLOPT_RETURNTRANSFER => true,
  CURLOPT_HEADER         => false,
  CURLOPT_ENCODING       => "",
  CURLOPT_USERAGENT      => "webio",
  CURLOPT_CONNECTTIMEOUT => 20,
  CURLOPT_TIMEOUT        => 20,
  CURLOPT_MAXREDIRS      => 1,
  CURLOPT_SSL_VERIFYHOST => false,
  CURLOPT_SSL_VERIFYPEER => false);
   curl_setopt_array( $ch, $options );
   $data = curl_exec($ch);
   if ($data==false)
  { die('ERROR '.$URL); }
  else
  { $http_status = curl_getinfo($ch, CURLINFO_HTTP_CODE);
    if ($http_status==200)
```

```
    { echo $data; }
    else
    { die('Not found '.$URL); }
  }
  curl_close($ch);
?>
```

REST - Representational State Transfer

With REST (Representational State Transfer), the Web IOs provide another web-based communication channel.

Communication takes place via Web IO specific HTTP requests via the HTTP or HTTPS port entered under Basic Settings >> Network >> Access for Web Services.

To be able to exchange REST data, access must first be activated via Communication Paths >> Rest.

If REST access is to be protected against unauthorised access, you have the option of activating digest authentication. The requests must then be made as user “admin” with the administrator password or as user “operator” with the user password.

For the Web-IO Digital and Analogue units, it can also be specified whether the outputs may be changed via REST.

For read access, REST uses the HTTP command GET.

The Web IO supports three formats for responses to REST requests:

- JSON
- XML
- Text

JSON

Read access

For read access REST uses the HTTP command GET.

The Web-IO supports three formats for replies to REST requests:

The format used for replies can be determined using the request. Using

```
http://<ip-adresse>/rest/json
```

for example opens the entire process image of the Web-IO in JSON format. The reply then looks as follows:


```
{
  "info" :
  {
    "request" : " / rest / json",
    "time" : "2016 - 09 - 09,
    09 : 42 : 54",
    "ip" : "10.40.22.227",
    "devicename" : "WEBIO - CAFE27"
  },
  "iostate" :
  {
    "input" : [
      {
        "number" : 0,
        "state" : 0
      },
      {
        "number" : 1,
        "state" : 0
      }
    ],
    "output" : [
      {
        "number" : 0,
        "state" : 0
      },
      {
        "number" : 1,
        "state" : 0
      }
    ],
    "counter" : [
      {
        "number" : 0,
        "state" : 0
      },
      {
        "number" : 1,
        "state" : 0
      }
    ]
  },
  "system" :
  {
    "time" :
    {
      "time" : "2016 - 09 - 09,
      09 : 42 : 54"
    },
    "diagnosis" : [
      {
        "time" : "06.09.2016 09 : 42 : 54",
        "msg" : "Gerätestatus : OK"
      }
    ],
    "diagarchive" : [
```

```
      {
        "time" : "06.09.2016 09 : 42 : 54",
        "msg" : "Gerätestatus : OK"
      }
    ]
  }
}
```

With the Web-IO Analogue, the answer would be structured as follows:

```
{
  "info":  {
    "request":  "/rest/json",
    "time":  "2020-01-27,10:37:23",
    "ip":  "10.40.22.13",
    "devicename": "WEBIO-0873E3"
  },
  "iostate":  {
    "output":[{
         "name":  "kanal 1",
         "number":0,
         "unit":  "mA",
         "value":  3.8
       }, {
         "name":  "Kanal 2",
         "number":1,
         "unit":  "mA",
         "value":  16.8
       }]
  },
  "system":{
    "time":  "2020-01-27,10:37:23",
    "diagnosis":  [{
         "time":  "27.01.2020 10:37:23",
         "msg": "Gerätestatus: OK"
       }],
    "diagarchive":  [{
         "time":  "27.01.2020 10:37:23",
         "msg": "Gerätestatus: OK"
       }]
  }
}
```

To query only individual areas or points, the query can be formulated in more detail. Here, for example, the query of the inputs of a Web-IO Digital:

```
http://<ip-adresse>/rest/json/iostate/input
```

This causes the Web IO to return the status of all inputs:

```
{
  "iostate" :
  {
    "input" : [
      {
        "number" : 0,
```

```
        "state" : 0
      },
      {
        "number" : 1,
        "state" : 0
      }
    ]
  }
}
```

With

```
http://<ip-adresse>/rest/json/iostate/input/0
```

the state of input 0 can be specifically queried.

```
{
  "iostate" :
  {
    "input" : [
      {
        "number" : 0,
        "state" : 0
      }
    ]
  }
}
```

Changing access

For accesses that change the switching state of the outputs or delete the counters, POST is used.

For example, to set Output 1 to ON, a POST is sent to the following URL:

```
http://<ip-adresse>/rest/json/iostate/output/1
```

The following parameters are passedas payload:

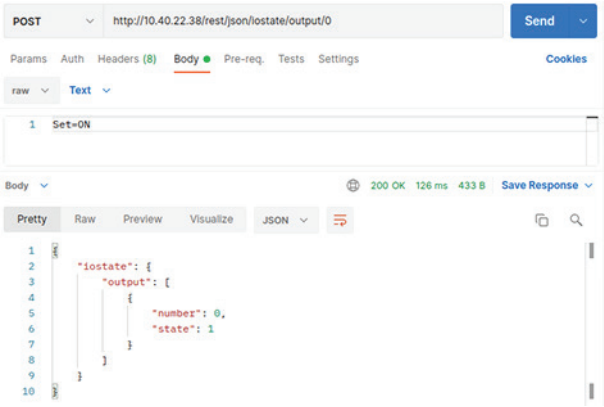
```
Set=ON
```

The Web IO responds with

```
{
  "iostate" :
  {
    "output" : [
      {
        "number" : 1,
```

```
        "state" : 1
      }
    ]
  }
}
```

The Web IO responds with



The output can be switched off via the same URL with the parameter `Set=OFF`.

The deletion, e.g. of Counter1, is done via a POST to the following URL:

```
http://<ip-adresse>/rest/json/iostate/counterclear/1
```

The Web IO responds with

```
{
  "iostate" :
  {
    "counter" : [
      {
        "number" : 1,
        "state" : 0
      }
    ]
  }
}
```

To get the answers in one of the other formats, simply replace the keyword json with xml or text.

XML

Reading access

With

```
http://<ip-adresse>/rest/xml
```

the entire process image of the Web IO can be retrieved in XML format. The response then looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<webio>
  <info>
    <request>/rest/xml</request>
    <time>2017-04-04,13:32:39</time>
    <ip>10.40.22.236</ip>
    <devicename>WEBIO-CAFE36</devicename>
  </info>
  <iostate>
       <input>
         <number>0</number>
         <state>1</state>
       </input>
       <input>
         <number>1</number>
         <state>0</state>
       </input>
       <output>
         <number>0</number>
         <state>0</state>
       </output>
       <output>
         <number>1</number>
         <state>0</state>
       </output>
       <counter>
         <number>0</number>
         <state>13136</state>
       </counter>
       <counter>
         <number>1</number>
         <state>2127</state>
       </counter>
  </iostate>
  <system>
    <time>2017-04-04,13:32:39</time>
      <diagnosis>
        <time>18.04.2017 13:32:39</time>
        <msg>Gerätestatus: OK</msg>
      </diagnosis>
      <diagarchive>
        <time>18.04.2017 13:32:39</time>
        <msg>Gerätestatus: OK</msg>
```

```
      </diagarchive>
  </system>
</webio>
```

To query only individual areas or points, the request can be formulated in more detail:

```
http://<ip-adresse>/rest/xml/iostate/input
```

This causes the Web IO to return the status of all inputs:

```
<?xml version="1.0" encoding="UTF-8"?>
<webio>
  <iostate>
      <input>
        <number>0</number>
        <state>1</state>
      </input>
      <input>
        <number>1</number>
        <state>0</state>
      </input>
  </iostate>
</webio>
```

Changing access

POST is used for accesses that change the switching state of the outputs or delete the counters.

For example, to set output 1 to ON, a POST is sent to the following URL:

```
http://<ip-adresse>/rest/xml/iostate/output/1
```

The following parameters are transferred as payload:

```
Set=ON
```

The Web IO responds with

```
<?xml version="1.0" encoding="UTF-8"?>
<webio>
  <iostate>
      <output>
        <number>1</number>
        <state>1</state>
      </output>
  </iostate>
</webio>
```

The output can be switched off via the same URL with the parameter `Set=OFF`.

The deletion, e.g. of Counter1, is done via a POST to the following URL:

```
http://<ip-adresse>/rest/xml/iostate/counterclear/1
```

The Web IO responds with

```
<?xml version="1.0" encoding="UTF-8"?>
<webio>
  <iostate>
      <counter>
        <number>1</number>
        <state>0</state>
      </counter>
  </iostate>
</webio>
```

Text

Reading access

Mit

```
http://<ip-adresse>/rest/text
```

the entire process image of the Web IO can be retrieved in Text format. The response then looks like this:

```
"info":"request":"/rest/text";                "time":"2017-04-04,13:43:01";
"ip":"10.40.22.236"; "devicename":"WEBIO-CAFE36";"iostate": : "number":0;
"state":0; : "number":0; "state":0; "number":1; "state":0; : "number":0;
"state":13612;  "number":1;  "state":2604;  "system":  "time":"2017-04-
04,13:43:01"; : "time":"18.04.2017 13:43:01"; "msg":"Gerätestatus: OK"; :
"time":"18.04.2017 13:43:01"; "msg":"Gerätestatus: OK";
```

To query only individual areas or points, the request can be formulated in more detail:

```
http://<ip-adresse>/rest/text/iostate/input
```

This causes the Web IO to return the status of all inputs:

```
"iostate": : "number":0; "state":0; "number":1; "state":0;
```

Modbus TCP - standardized access

Modbus TCP is a software interface for address-based access to process data. The W&T Web-IOs when appropriately configured act as Modbus servers which can be controlled by Modbus controllers (clients or masters).

Programmierer, die den Modbus-Zugang in eigenen Applikationen nutzen möchten, finden hier eine detaillierte Beschreibung zu den vom Web-IO unterstützten Register und Funktionscodes.

Using Modbus TCP access you can read the status of outputs, inputs and counters. state and system polling is also possible. In addition the outputs can be switched via Modbus TCP and the counters cleared.

The Web-IO provides 64 16-bit registers which can be freely written and which can be accessed from the browser.

Unter *Kommunikationswege >> Modbus-TCP* kann der Modbus-Zugang des Web-IO aktiviert werden. Von Hause aus ist der lokale TCP-Port, wie für Modbus-TCP üblich auf 502 voreingestellt.

Outputs die über Modbus-TCP gesteuert werden sollen, können über *Outputs für Modbus/TCP freigeben* aktiviert werden.

Modbus TCP communication

Modbus TCP is a master/slave procedure in which a Modbus master sends a request to the slave (Web-IO) and the slave (Web-IO) answers with a reply.

Data exchange between the Modbus client and the Web-IO takes place using Modbus TCP packets.

The Web-IO has a memory from which the Modbus master can read or to which it can write. This memory is divided into areas which represent certain characteristics of the Web-IO.

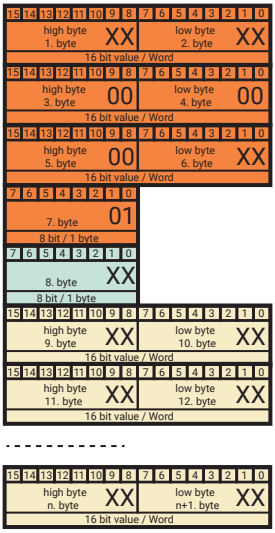
There are areas for the status of inputs, outputs and counters. Other areas indicate the system status or pertain to the alarms.

A special feature of the Web-IO is a memory area which on the one hand can be freely read or written via Modbus access and on the other hand can be called via the Web interface. With appropriate Web page programming a Modbus system can therefore be used for visualizing process data in the browser.

Another special function of the Web-IO is that an alarm can be triggered by writing to a particular memory area.

A list with the exact memory organization follows later in this section.

The Modbus data packets always consist of a header, the function code, the start address and other parameters and registers.



Transaction ID	16 bit Is incremented by 1 per reqest by the
Protocol ID	16 bit always = 0x0000
Length	16 bit Number of bytes following this field
Unit ID	8 bit always = 0x0001
Function Code	8 bit Determines the purpose of the request
Start Address	16 bit Address of the registers to be read/write
More Parameters or register	8 bit, 16 bit, 32 bit

Transaction Identifier

Used to classify the reply from the Web-IO to the request of the client. The client normally increments the ID by 1 with each sending of data. The Web-IO always returns the received value 1:1.

Protocol Identifier

Has no meaning for communication with the Web-IO and is always 0x0000

Length
Number of bytes sent by *Length* (entire number of bytes sent - 6).

Unit identifier
Always 0x01 for Modbus TCP

Function Code
The *Function Code* specifies how the Modbus memory of the Web-IO is accessed:

- By bit
- By register (16-bits)

and what the purpose of the request is:

- 0x01 Read Coils - read individual bits
- 0x02 Read Discrete Inputs - read individual bits
- 0x03 Read Holding Registers- read multiple registers
- 0x04 Read Input Register - read multiple input registers.
- 0x05 Write Single Coil - write a bit
- 0x06 Write Single Register - write only one register
- 0x07 Read Exception State - read error status
- 0x0F Write Multiple Coils - write multiple bits
- 0x10 Write Multiple Register - write multiple registers

Start Address
Start Address specifies which area of the Modbus memory in the Web-IO will be accessed. This also determines which characteristic is accessed (inputs, outputs, counters, alarms, ...).

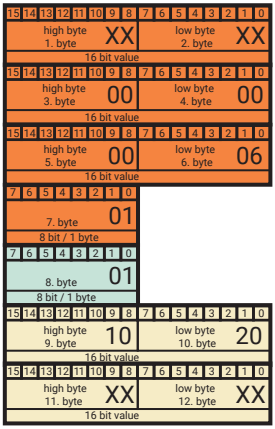
A list with the exact memory organization can be found later in this section.

Both the **Function Code** *and* **Start Address** *determine what the sent request should trigger for the Web-IO.*

Other parameters and registers
Depending on which *Function Code* is used other parameters and registers may follow.

Function Code 0x01 Read Coils

Function Code 0x01 is intended for binary reading of the status of the Web-IO outputs.



Transaction ID	
Protocol ID	
Length	
Unit ID	
Function Code	8 bit 0x01 Read Coil
Start Address	16 bit Address of the registers to be read
Quantity of Outputs	16 bit Number of inputs to be read

Start Address

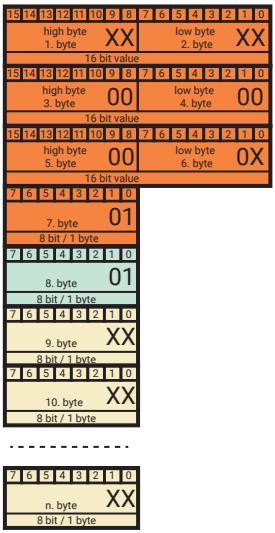
Specifies starting at which address bits (outputs) should be read.

Quantity of Outputs

Here the number of bits (outputs) to be read is transmitted.

By sending a data packet with FC 0x01 one or more Boolean values (output states, i.e. 0/1 or ON/OFF) can be requested.

The Web-IO replies as follows:



Transaction ID	
Protocol ID	
Length	
Unit ID	
Function Code	
byte Count	8 bit Number of bytes returned (one byte for every 8 bits or part there of)
Output State	8 bit The requested bits are passed starting at the with the 10th byte bit 0

Bytes Count
Contains the number of send bytes which contain the requested bits. One byte per 8 started bits is sent. That means: starting with the 9th bit 2 bytes are sent.

Output State
Number of bytes, as sent in *Byte Count*. The first State Byte (10th byte) contains, beginning with Bit 0, the first 8 requested bits (outputs). If less then 8 bits were requested, the unused bytes are sent with 0. For more than 8 bits it continues with the 11th byte.

Example: Read the status of Outputs 0 and 1
Output 0 is OFF, Output 1 is ON

Start Address is set to 0x1020, *Quantity of Outputs* to 0x02. (The outputs can be called starting at address 0x1020.)

The Web-IO replies with *Byte Count* = 1, and *Output State* = 0x02.

Function Code 0x02 Read Discrete Inputs

Function Code 0x02 is intended for binary reading of the status of Web-IO inputs.

The packet construction and reply with *Function Code* 0x02 is identical to that of Function Code 0x01.

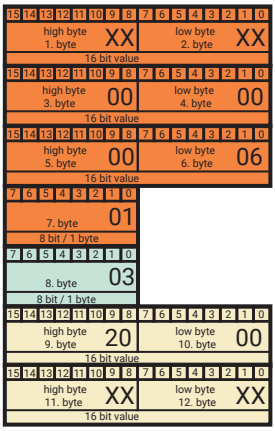
Example: Read the status of Inputs 0 and 1
Input 0 is OFF, Input1 is ON

Start Address is set to 1000, *Quantity of Inputs* to 2.
(The inputs can be called starting at address 0x1000.)

The Web-IO replies with *Byte Count* = 1, and *Output State* = 0x02.

Function Code 0x03 Read Holding Registers

Function Code 0x03 is intended for reading multiple registers (16-bit values). Using FC 0x03 the values of inputs, outputs, counters etc. can be polled for the Web-IO depending on which *Start Address* is used.

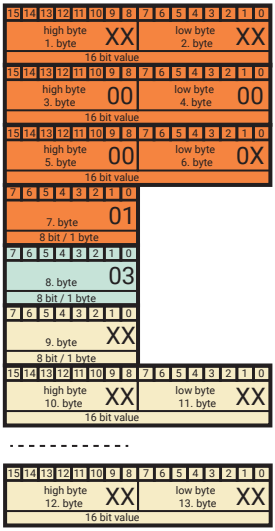


Transaction ID	
Protocol ID	
Length	
Unit ID	
Function Code	
Start Address	16 bit Address of the registers to be read
Number of Registers	16 bit Number of registers to be read

Start Address
Specifies the address to start reading the registers.

Number of Registers
Here the number of registers to read is transmitted.

The Web-IO replies with the following packet:



Transaction ID

Protocol ID

Length

Unit ID

Function Code

byte Count — 8 bit
Number of bytes returned
(2 bytes per register)

Output State — 16 bit
The requested registers are transferred starting with the 10th byte.

Bytes Count
Contains the number of bytes sent as registers (2 bytes per 16-bit register).

Register Value
One or more 16-bit register values. The first 16-bit register begins with the high byte at the position of the 10th byte.

Depending on the start address (beginning at 0x5000) two 16-bit values - i.e. 4 bytes - are passed for one requested 32-bit register. In this case again the value begins with the highest byte at the position of the 10th byte and the first low byte lies at the position of the 13th byte of the data packet.

Example:
Read the status of the outputs of a Web-IO 2xDigital
Output 0 is OFF, Output 1 is ON

Start Address is set to 0x2002,
Number of Registers to 0x01
(The outputs can be called as registers starting at address 0x2002.)

The Web-IO replies with

Byte Count = 0x02,
Register Value = 0x0002.

Function Code 0x04 Read Input Registers

Function Code 0x04 is especially provided for reading the status of the Web-IO inputs as 16-bit registers.

The packet structure of request and reply with Function Code 0x04 is identical to that of Function Code 0x03.

Example:
Read the status of the inputs of a Web-IO 2xDigital
Input 0 is OFF, Input1 is ON

Start Address is set to 0x2000,
Number of Registers to 0x01
(The inputs can be called as registers starting at address 0x2002.)

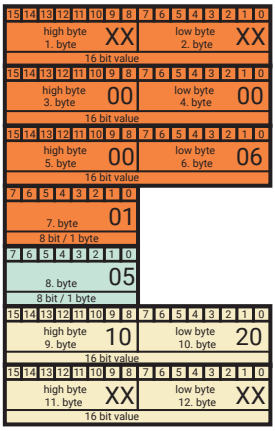
The Web-IO replies with

Byte Count = 0x02

Register Value = 0x0002

Function Code 0x05 Write Single Coil

Function Code 0x05 is intended for setting a single output.



Transaction ID

Protocol ID

Length

Unit ID

Function Code

Register Address 16 bit
Address of the registers to be write

Output Value 16 bit
Value to be written into the register

Output Address

Using *Output Address* you indirectly determine which output to switch. Beginning at 0x1020 each output has its own address.

Output Value

Here you specify whether an output should be turned ON or OFF:

ON = 0xFF00
OFF = 0x0000

The Web-IO replies with a data packet having the exact same structure.

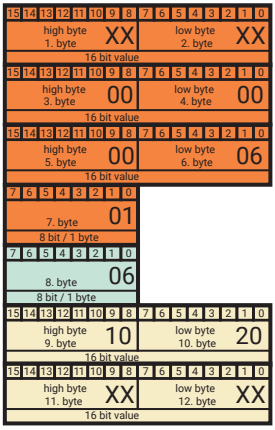
Example:
Output 1 of a Web-IO should be turned ON
Output Address is set to 0x1021,
Output Value to 0xFF00

The Web-IO replies with

Output Address = 1021
Output Value = 0xFF00

Function Code 0x06 Write Single Register

Function Code 0x06 works just the same way as Function Code 0x05 and is used for setting any desired register.



Transaction ID

Protocol ID

Length

Unit ID

Function Code

Register Address 16 bit
Address of the registers to be write

Output Value 16 bit
Value to be written into the register

Register Address

Register Address specifies which register address to write to.

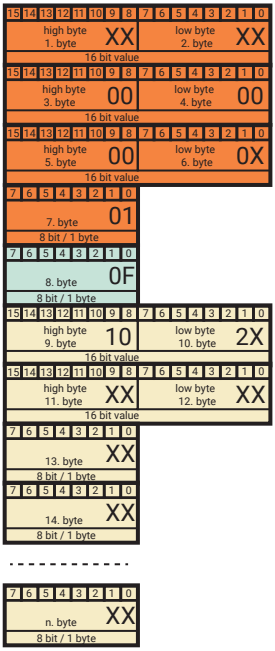
Register Value

A 16-bit register value which is written to the Modbus memory of the Web-IO.

The Web-IO replies with a data packet having the exact same structure.

Function Code 0x0F Write Multiple Coils

Function Code 0x0F is intended for bit setting of outputs.



Transaction ID	
Protocol ID	
Length	
Unit ID	
Function Code	
Start Address	16 bit Address of the first output to be set
Quantity of Outputs	16 bit Number of outputs to be set
byte Count	8 bit Number of bytes returned (one byte for every 8 bits or part thereof)
Output State	8 bit The requested bits are passed starting at the with the 14th byte bit 0

Start Address

Specifies at which address to begin writing output bits.

Quantity of Outputs

Here the number of bits to write is transmitted.

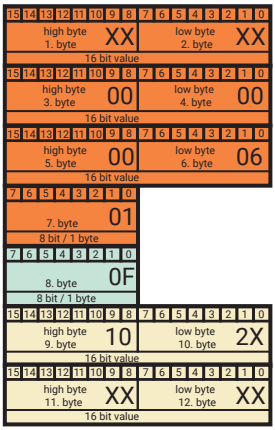
Bytes Count

Indicates the number of transmitted bytes which the output bits being set contain. One byte per 8 bits begun is sent. This means 2 bytes are sent beginning at the 9th bit.

Output State

Number of bytes as sent in *Byte Count*. The first byte (14th byte) contains, beginning at Bit 0, the first 8 output bits (outputs) to be set. If fewer than bits (outputs) were specified, the unused bits are sent with 0. For more than 8 bits it continues with the 15th byte.

The Web-IO replies with a data packet having the following structure:



Transaction ID

Protocol ID

Length

Unit ID

Function Code

Start Address — 16 bit
Address of the first output to be set

Quantity of Outputs — 16 bit
Number of outputs to be set

Start Address
Specifies at which address to begin writing output bits.

Quantity of Outputs
Here the number of output bits written is sent.

Example:
Output 1 on a Web-IO should be turned ON
Output 0 should be turned OFF, Output 1 is ON

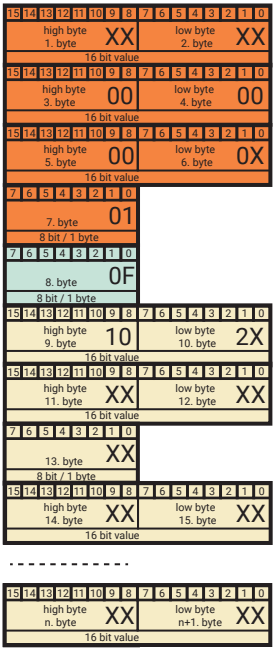
Output Address is set to 1020,
Quantity of Outputs to 0x02
Byte Count to 0x01
One byte Output Value with content 0x02 is sent.

The Web-O replies with

Output Address = 1020
Quantity of Outputs = 0x0002

Function Code 0x10 Write Multiple Registers

Function Code 0x0F is intended for writing multiple 16-bit register values.



Transaction ID	
Protocol ID	
Length	
Unit ID	
Function Code	
Start Address	16 bit Address of the first register to be set
Quantity of Registers	16 bit Number of registers to write
byte Count	8 bit 2 bytes are counted for each 16-bit register
Register Value	16 bit Values to write

Start Address

Specifies at which address to begin writing registers.

Quantity of Registers

Here the number of 16-bit registers to write is sent. When writing to the 32-bit area of the Web-IO two 16-bit registers must be counted per 32-bit value.

Bytes Count

Contains the number of bytes to be sent. 2 bytes are counted for each 16-bit register to be sent.

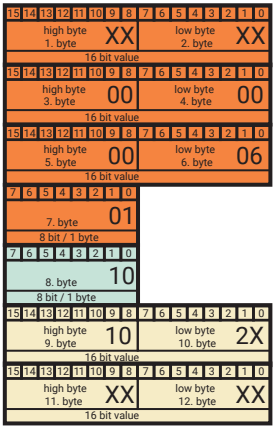
Register Value

Here the 16-bit registers are sent. The first high byte is set as the 14th byte in the data packet, the first low byte as the 15th byte, etc.

When writing to the 32-bit area of the Web-IO (starting at address 0x5000) two 16-

bit registers per 32-bit value beginning with the highest byte of the 32-bit value must be written.

The Web-IO replies with a data packet having the following structure:



Transaction ID

Protocol ID

Length

Unit ID

Function Code

Start Address — 16 high byte
Address of the first Register

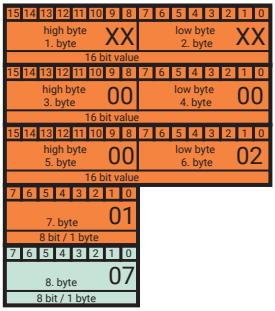
Quantity of Registers — 16 high byte
Quantity of Registers set

Start Address
Specifies at which address to begin writing the 16-bit registers.

Quantity of Outputs
Here the number of written 16-bit registers is sent.

Function Code 0x07 Read Exception State

Function Code 0x07 is intended for reading the exception state (error status) of the Web-IO.



Transaction ID

Protocol ID

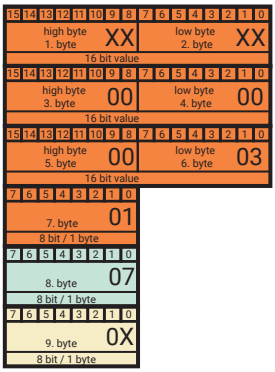
Length

Unit ID

Function Code

Packets set by the master with *Function Code* 0x07 consist only of a Modbus header and *Function Code*. No other parameters are sent.

The Web-IO replies with a data packet having the following structure:



Transaction ID

Protocol ID

Length

Unit ID

Function Code

Exception Code

Exception Code
For the Web-IO the Exception Code simply indicates the state of the LEDs for *System Error* and *On Error*.

Bit 0 = 1 corresponds to *System Error*
Bit 2 = 1 corresponds to *ON Error*

All other bits have no meaning for the Web-IO and are sent as 0.

Incorrect master requests

If the Modbus TCP master is supposed to send requests to the Web-IO which cannot be processed by the Web-IO in terms of form and number of parameters or with respect to the selected register address, the Web-IO returns a Modbus packet whose structure corresponds to a reply to *Function Code* 0x07.

In such cases the Web-IO adds 0x80 to the received *Function Code* and sets this code into its own packet.

The *Exception Code* includes additional details about the error

0x01 Function Code not supported by Web-IO
0x02 Write attempt to read-only area

0x03 Write attempt with wrong data
0x04 Write attempt to allowed area failed

Modbus address ranges for Web-IO Digital

The Web-IO Analog has different Modbus memory areas:

- Bit area from address 0x1000,
- 16Bit area from address 0x2000
- 32-Bit area bzw. 2x16-Bit area from address 0x5000

Addressing in the bit range is done bit by bit, i.e. 1 bit requires an address. In the 16-bit and 32-bit range, addressing takes place word by word (2 bytes).

The Inputs
can be found in the

- Bit area from address 0x1000
  (example 57737: 0x1000 und 0x1001),
- 16Bit area from address 0x2000 in or in the 16-Bit area from 0x2000
  (example 57737: 0x2000),
- 32-Bit area from 0x5000.

The Outputs
can be found in the

- Bit area from address 0x1020
  (example 57737: 0x1020 und 0x1021)
- 16-Bit area from address 0x2002.
- 32-Bit area from address 0x5002.

The Alarms
can be found in the

- Bit area from address 0x1040
  (example 57737: 0x1040 und 0x1041),
- area from address 0x2004.

Alarm triggers are in the bit range from address 0x1800.

The Counter
can be found in the

- 32-Bit area from address 0x5004
  (example 57737: 0x5004/0x5005 and 0x5006/0x5007).

Write counter values: Counters can be set to any value.

The Exception state
can be found in the

- Bit area from address 0x1050,
- 16-Bit area from address 0x200C (Low Byte).

Alternatively, the exception status is read out with the function code 0x07.

The configuration status
can be found in the

- Bit area from address 0x1058,
- 16-Bit area from address 0x200C (High Byte).

Der Diagnose-state
(Number of errors) can be found in the

16Bit area from address 0x2005,
im 32-Bit area from address 0x5048.

The diagnostic status bits
can be found in the

- 16Bit area from address 0x2006,
- 32-Bit area from address 0x5048.

Device identification
via serial number (from 0x6000) and Mac address (from 0x6004).

Modbus memory layout

Bit area:

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read bit with FC (decimal)	Read bit with FC (hexadecimal)	Write bit with FC (decimal)	Write bit with FC (hexadecimal)
Input 0	4096	0x1000	Bit	1	1, 2	0x01, 0x02		
Input 1	4097	0x1001	Bit	1	1, 2	0x01, 0x02		
Input 2	4098	0x1002	Bit	1	1, 2	0x01, 0x02		
Input 3	4099	0x1003	Bit	1	1, 2	0x01, 0x02		
Input 4	4100	0x1004	Bit	1	1, 2	0x01, 0x02		
Input 5	4101	0x1005	Bit	1	1, 2	0x01, 0x02		
Input 6	4102	0x1006	Bit	1	1, 2	0x01, 0x02		
Input 7	4103	0x1007	Bit	1	1, 2	0x01, 0x02		
Input 8	4104	0x1008	Bit	1	1, 2	0x01, 0x02		
Input 9	4105	0x1009	Bit	1	1, 2	0x01, 0x02		
Input 10	4106	0x100A	Bit	1	1, 2	0x01, 0x02		
Input 11	4107	0x100B	Bit	1	1, 2	0x01, 0x02		
Input 12	4108	0x100C	Bit	1	1, 2	0x01, 0x02		
Input 13	4109	0x100D	Bit	1	1, 2	0x01, 0x02		
Input 14	4110	0x100E	Bit	1	1, 2	0x01, 0x02		
Input 15	4111	0x100F	Bit	1	1, 2	0x01, 0x02		
Output 0	4128	0x1020	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 1	4129	0x1021	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 2	4130	0x1022	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 3	4131	0x1023	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 4	4132	0x1024	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 5	4133	0x1025	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 6	4134	0x1026	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 7	4135	0x1027	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 8	4136	0x1028	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 9	4137	0x1029	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 10	4138	0x102A	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 11	4139	0x102B	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 12	4140	0x102C	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 13	4141	0x102D	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 14	4142	0x102E	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Output 15	4143	0x102F	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Action 1 state	4160	0x1040	Bit	1	1, 2	0x01, 0x02		
Action 2 state	4161	0x1041	Bit	1	1, 2	0x01, 0x02		
Action 3 state	4162	0x1042	Bit	1	1, 2	0x01, 0x02		
.....								
Action 28 state	4187	0x105B	Bit	1	1, 2	0x01, 0x02		

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read bit with FC (decimal)	Read bit with FC (hexadecimal)	Write bit with FC (decimal)	Write bit with FC (hexadecimal)
Action 29 state	4188	0x105C	Bit	1	1, 2	0x01, 0x02		
Action 30 state	4189	0x105D	Bit	1	1, 2	0x01, 0x02		
Exception state	4192	0x1060	Bit	1	1, 2	0x01, 0x02		
Config. state	4200	0x1068	Bit	1	1, 2	0x01, 0x02		
Action 1 Trigger	4160	0x1800	Bit	1	1, 2	0x01, 0x02		
Action 2 Trigger	4161	0x1801	Bit	1	1, 2	0x01, 0x02		
Action 3 Trigger	4162	0x1802	Bit	1	1, 2	0x01, 0x02		
.....								
Action 28 Trigger	4187	0x105B	Bit	1	1, 2	0x01, 0x02		
Action 29 Trigger	4188	0x105C	Bit	1	1, 2	0x01, 0x02		
Action 30 Trigger	4189	0x105D	Bit	1	1, 2	0x01, 0x02		

16-Bit area:

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read register with FC (decimal)	Read register with FC (hexadecimal)	Write register with FC (decimal)	Write register with FC (hexadecimal)
Inputs 0 - 15	8192	0x2000	16-Bit	2	3, 4	0x03, 0x04		
Outputs 0 - 15	8194	0x2002	16-Bit	2	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Action state 1 - 16	8196	0x2004	16-Bit	2	3, 4	0x03, 0x04		
Diagn. Error count	8198	0x2006	16-Bit	2	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Diag. state 0 - 15	8199	0x2007	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 16 - 31	8200	0x2008	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 32 - 47	8201	0x2009	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 48 - 63	8202	0x200A	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 64 - 79	8203	0x200B	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 80 - 95	8204	0x200C	16-Bit	2	3, 4	0x03, 0x04		
Except./Conf.-State	8205	0x200D	16-Bit	2	3, 4	0x03, 0x04		

32-Bit area:

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read register with FC (decimal)	Read register with FC (hexadecimal)	Write register with FC (decimal)	Write register with FC (hexadecimal)
Inputs 0 - 15	20480	0x5000	32-Bit	4	3, 4	0x03, 0x04		
Outputs 0 - 15	20482	0x5002	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Action State 1 - 30	20484	0x5004	32-Bit	4	3, 4	0x03, 0x04		
Counter 0	20486	0x5006	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 1	20488	0x5008	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 2	20490	0x500A	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 3	20492	0x500C	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 4	20494	0x500E	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 5	20496	0x5010	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 6	20498	0x5012	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 7	20500	0x5014	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 8	20502	0x5016	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 9	20504	0x5018	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 10	20506	0x501A	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 11	20508	0x501C	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 12	20510	0x501E	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 13	20512	0x5020	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 14	20514	0x5022	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Counter 15	20516	0x5024	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Diag. Error count	20554	0x504A	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Diag. state 0 - 31	20556	0x504C	32-Bit	4	3, 4	0x03, 0x04		
Diag. state 32 - 63	20558	0x504E	32-Bit	4	3, 4	0x03, 0x04		
Diag. state 64 - 95	20560	0x5050	32-Bit	4	3, 4	0x03, 0x04		
Diag state 96 - 127	20562	0x5052	32-Bit	4	3, 4	0x03, 0x04		
D. state 128 - 159	20564	0x5054	32-Bit	4	3, 4	0x03, 0x04		
Diag state 160 - 191	20566	0x5056	32-Bit	4	3, 4	0x03, 0x04		
Serial number	24574	0x6000	32-Bit	8	3, 4	0x03, 0x04		
Ethernet-Address	24578	0x6004	32-Bit	8	3, 4	0x03, 0x04		

Please note that, depending on the Web-IO used, the full range of inputs, outputs, counters or alarms may not be available.

Modbus address ranges for Web-IO Analog

The Web-IO Analog has different Modbus memory areas:

- Bit area from address 0x1000,

- 16-Bit area from address ab Adresse 0x2000
- 32-Bit area from address e.g.. 2x16 area from address 0x5000)

Addressing in the bit range is done bit by bit, i.e. 1 bit requires an address. In the 16-bit and 32-bit range, addressing takes place word by word (2 bytes).

The Inputs and Outputs
can be found in the 32area

- Inputs from address 0x5036.
- Outputs from address 0x5046.

The values are given as percentages with 3 decimal digits and are to be understood as relative.

Value	dec	hex	4-20mA	0-10V
0%	0	0x0000 0000	4	0
0,1%	10	0x0000 000A	4,0016	0,001
1%	1.000	0x0000 03E8	4,16	0,1
10%	10.000	0x0000 2710	5,6	1
65,5%	65.535	0x0000 FFFF	14,4856	6,55
65,5%	65.536	0x0001 0000	14,4857	6,55
100%	100.000	0x0001 86A0	20	10
120%	120.000	0x0001 D4C0	23,2	12

The Alarms
can be found in the

- Bit area from address 0x1040
- 16-Bit area from address 0x2004.

Alarm triggers are in the bit range from address 0x1800.

The Exception state
can be found in the

- Bit area from address 0x1050,
- 16-Bit area from address 0x200C (Low Byte).

Alternatively, the exception state is read out with the function code 0x07.

The configuration status
can be found in the

- Bit area from address 0x1058,
- 16-Bit area from address 0x200C (High Byte).

Der Diagnose-state
(Number of errors) can be found in the

16Bit area from address 0x2005,
im 32-Bit area from address 0x5048.

The diagnostic status bits
can be found in the

- 16Bit area from address 0x2006,
- 32-Bit area from address 0x5048.

Device identification
via serial number (from 0x6000) and Mac address (from 0x6004).

Bit area:

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read bit with FC (decimal)	Read bit with FC (hexadecimal)	Write bit with FC (decimal)	Write bit with FC (hexadecimal)
Action 1 state	4160	0x1040	Bit	1	1, 2	0x01, 0x02		
Action 2 state	4161	0x1041	Bit	1	1, 2	0x01, 0x02		
Action 3 state	4162	0x1042	Bit	1	1, 2	0x01, 0x02		
.....								
Action 28 state	4187	0x105B	Bit	1	1, 2	0x01, 0x02		
Action 29 state	4188	0x105C	Bit	1	1, 2	0x01, 0x02		
Action 30 state	4189	0x105D	Bit	1	1, 2	0x01, 0x02		
Exception State	4192	0x1060	Bit	1	1, 2	0x01, 0x02		
Config. state	4200	0x1068	Bit	1	1, 2	0x01, 0x02		
Action 1 Trigger	4160	0x1800	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Action 2 Trigger	4161	0x1801	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Action 3 Trigger	4162	0x1802	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
.....							5, 15	0x05, 0x0F
Action 28 Trigger	4187	0x105B	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Action 29 Trigger	4188	0x105C	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F
Action 30 Trigger	4189	0x105D	Bit	1	1, 2	0x01, 0x02	5, 15	0x05, 0x0F

16-Bit area:

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read register with FC (decimal)	Read register with FC (hexadecimal)	Write register with FC (decimal)	Write register with FC (hexadecimal)
Action state 1 - 16	9196	0x2004	16-Bit	2	3, 4	0x03, 0x04		
Diagn. Error count	9198	0x2006	16-Bit	2	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Diag. state 0 - 15	9199	0x2007	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 16 - 31	9200	0x2008	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 32 - 47	9201	0x2009	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 48 - 63	9202	0x200A	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 64 - 79	9203	0x200B	16-Bit	2	3, 4	0x03, 0x04		
Diagn.state 80 - 95	9204	0x200C	16-Bit	2	3, 4	0x03, 0x04		
Except/Conf.-state	9205	0x200D	16-Bit	2	3, 4	0x03, 0x04		

32-Bit area:

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read register with FC (decimal)	Read register with FC (hexadecimal)	Write register with FC (decimal)	Write register with FC (hexadecimal)
Action state 1 - 30	20484	0x5004	32-Bit	4	3, 4	0x03, 0x04		
Analog In 1	20534	0x5036	32-Bit	4	3, 4	0x03, 0x04		
Analog In 2	20536	0x5038	32-Bit	4	3, 4	0x03, 0x04		
Analog Out 1	20550	0x5046	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Analog Out 2	20552	0x5048	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Diagn. Error Count	20554	0x504A	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Diag. state 0 - 31	20556	0x504C	32-Bit	4	3, 4	0x03, 0x04		
Diag. state 32 - 63	20558	0x504E	32-Bit	4	3, 4	0x03, 0x04		
Diag. state 64 - 95	20560	0x5050	32-Bit	4	3, 4	0x03, 0x04		
D. state 96 - 127	20562	0x5052	32-Bit	4	3, 4	0x03, 0x04		
D. state 128 - 159	20564	0x5054	32-Bit	4	3, 4	0x03, 0x04		
Diag state 160 - 191	20566	0x5056	32-Bit	4	3, 4	0x03, 0x04		
Serial number	24574	0x6000	32-Bit	8	3, 4	0x03, 0x04		
Ethernet-Address	24578	0x6004	32-Bit	8	3, 4	0x03, 0x04		

Bitte beachten Sie, dass je nach verwendetem Web-IO nicht die gesamte Breite der Inuts, Outputs, Counter oder Alarme zur Verfügung steht.

Modbus address ranges for Web-Thermometer

There are different Modbus memory areas for the Web Thermometers, the Web Thermo-Hygrometer and the Web Thermo-Hygrobarometer:

- 16-Bit area from address ab Adresse 0x2000
- 32-Bit area from address e.g.. 2x16 area from address 0x5000)

In the 16-bit and 32-bit range, addressing takes place word by word (2 bytes).

Die Messwerte
can be found in the 32-Bit area from address 0x5036.

The values are output in 1/10°C (or 1/10%rH or 1/10hPa). The MSB determines whether the value is positive (0) or negative (1).

The Alarms
can be found in the

- Bit area from address 0x1040
- 16-Bit area from address 0x2004.

Alarm triggers are in the bit range from address 0x1800.

The Exception state
can be found in the

- Bit area from address 0x1050,
- 16-Bit area from address 0x200C (Low Byte).

Alternatively, the exception state is read out with the function code 0x07.

The configuration status
can be found in the

- Bit area from address 0x1058,
- 16-Bit area from address 0x200C (High Byte).

Der Diagnose-state
(Number of errors) can be found in the

16Bit area from address 0x2005,
im 32-Bit area from address 0x5048.

The diagnostic status bits
can be found in the

- 16Bit area from address 0x2006,
- 32-Bit area from address 0x5048.

Device identification
via serial number (from 0x6000) and Mac address (from 0x6004).

16-Bereich:

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read register with FC (decimal)	Read register with FC (hexadecimal)	Write register with FC (decimal)	Write register with FC (hexadecimal)
Action state 1 - 16	9196	0x2004	16-Bit	2	3, 4	0x03, 0x04		
Diagn. Error count	9198	0x2006	16-Bit	2	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Diag. state 0 - 15	9199	0x2007	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 16 - 31	9200	0x2008	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 32 - 47	9201	0x2009	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 48 - 63	9202	0x200A	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 64 - 79	9203	0x200B	16-Bit	2	3, 4	0x03, 0x04		
Diag. state 80 - 95	9204	0x200C	16-Bit	2	3, 4	0x03, 0x04		
Except/Conf.-state	9205	0x200D	16-Bit	2	3, 4	0x03, 0x04		

32-Bit-Bereich:

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read register with FC (decimal)	Read register with FC (hexadecimal)	Write register with FC (decimal)	Write register with FC (hexadecimal)
Action state 1 - 30	20484	0x5004	32-Bit	4	3, 4	0x03, 0x04		
Measurement 1	20534	0x5036	32-Bit	4	3, 4	0x03, 0x04		
Measurement 2	20536	0x5038	32-Bit	4	3, 4	0x03, 0x04		
Measurement 3	20550	0x503A	32-Bit	4	3, 4	0x03, 0x04		
Measurement 4	20552	0x503C	32-Bit	4	3, 4	0x03, 0x04		
Measurement 5	20534	0x503E	32-Bit	4	3, 4	0x03, 0x04		
Measurement 6	20536	0x5040	32-Bit	4	3, 4	0x03, 0x04		

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read register with FC (decimal)	Read register with FC (hexadecimal)	Write register with FC (decimal)	Write register with FC (hexadecimal)
Measurement 7	20550	0x5042	32-Bit	4	3, 4	0x03, 0x04		
Measurement 8	20552	0x5044	32-Bit	4	3, 4	0x03, 0x04		
Diagn. Error Count	20554	0x504A	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Diag. state 0 - 31	20556	0x504C	32-Bit	4	3, 4	0x03, 0x04		
Diag. state 32 - 63	20558	0x504E	32-Bit	4	3, 4	0x03, 0x04		
Diag. state 64 - 95	20560	0x5050	32-Bit	4	3, 4	0x03, 0x04		
D. state 96 - 127	20562	0x5052	32-Bit	4	3, 4	0x03, 0x04		
D. state 128 - 159	20564	0x5054	32-Bit	4	3, 4	0x03, 0x04		
Diag state 160 - 191	20566	0x5056	32-Bit	4	3, 4	0x03, 0x04		
Serial number	24574	0x6000	32-Bit	8	3, 4	0x03, 0x04		
Ethernet-Address	24578	0x6004	32-Bit	8	3, 4	0x03, 0x04		

Virtual registers for browser interaction

All W&T devices offer a freely writable address range for data exchange with web applications. The memory area that the device makes available for transferring values is from address 0x7000.

32-Bit area:

Description	Address (decimal)	Address (hexadecimal)	Storage model	Length (Byte)	Read register with FC (decimal)	Read register with FC (hexadecimal)	Write register with FC (decimal)	Write register with FC (hexadecimal)
Virtual Register 0	28672	0x7000	32-Bit	4	3, 4	0x03, 0x04	6, 16	0x06, 0x10
Virtual Register 1	28674	0x7002	32-Bit		3, 4	0x03, 0x04	6, 16	0x06, 0x10
Virtual Register 2	28676	0x7004	32-Bit		3, 4	0x03, 0x04	6, 16	0x06, 0x10
Virtual Register 3	28678	0x7006	32-Bit		3, 4	0x03, 0x04	6, 16	0x06, 0x10
Virtual Register 4	28680	0x7008	32-Bit		3, 4	0x03, 0x04	6, 16	0x06, 0x10
Virtual Register 5	28682	0x700A	32-Bit		3, 4	0x03, 0x04	6, 16	0x06, 0x10
Virtual Register ..								
Virtual Register 28	28728	0x7038	32-Bit		3, 4	0x03, 0x04	6, 16	0x06, 0x10
Virtual Register 29	28730	0x703A	32-Bit		3, 4	0x03, 0x04	6, 16	0x06, 0x10
Virtual Register 30	28732	0x703C	32-Bit		3, 4	0x03, 0x04	6, 16	0x06, 0x10
Virtual Register 31	28734	0x703E	32-Bit		3, 4	0x03, 0x04	6, 16	0x06, 0x10

Reading non-supported registers

When reading data (memory areas) which were not defined for the device the unit returns „0“.

Modbus - virtual registers

The Web-IO provides 64 virtual 16-bit registers to which the Modbus master can write any desired values (High Byte first). Writing to these registers triggers no special actions in the Web-IO.

The virtual memory is used rather for sending Modbus process data to Web applications.

Using an HTTP Request

```
modbusreg?PW=<password>&
```

the 64 registers (128 bytes) can be called by Web applications.

The Web-IO replies with

```
modbus;<High Byte1>;<Low Byte1>;<High Byte2>;<Low Byte2>;<High
Byte3>;......
```

In other words, all 64 registers (128 bytes) are output byte for byte separated by semicolons behind the word „modbus“.

Using JavaScript and programming techniques such as AJAX you can thereby implement process visualization in the browser.

In the simplest case the virtual registers can be displayed on the factory-side User page of the Web-IO (only if Modbus mode has been enabled in the Web-IO).

Web-IO User Site
WEBIO-05372A
Von der Klemme direkt aufs Netzwerk

Logout

Inputs				Outputs			
Name	*State*	*Counter*	*Clear*	*Name*	*State*	*Switch*	
Input 0	ON	0	Clear	Output 0	OFF	ON	OFF
Input 1	ON	0	Clear	Output 1	OFF	ON	OFF

Free available Modbus Memory

Addr.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
7000	26 38	FE 12	AF 45	20 16	39 45	FC D3	09 CD	23 FA	16 29	F3 D7	FD 2A	45 9F	35 2F	DA 29	58 39	F2 D1
7010	16 29	F3 D7	FD 2A	45 9F	35 2F	DA 29	58 39	F2 D1	23 4E	FC 56	01 29	56 39	23 E4	F1 AC	CD CD	45 91
7020	23 4E	FC 56	01 29	56 39	23 E4	F1 AC	CD CD	45 91	12 AC	56 FF	92 73	5F 2A	FA DE	34 1D	00 02	1F A1
7030	12 AC	56 FF	92 73	5F 2A	FA DE	34 1D	00 02	1F A1	00 00	00 00	00 00	00 00	00 00	00 00	00 00	00 00

Show as: ○ 8Bit ◉ 16Bit ○ 32Bit

Wiesemann & Theis GmbH
Porschestraße 12
D-42279 Wuppertal

Mail info@wut.de
Web www.wut.de

Tel. +49 202 2680-110
Fax +49 202 2680-265