SAUTER
Creating Sustainable Environments.

Product Manual

UNIVERSAL

Supports main building automation and IoT protocols. New protocols are added on customer request within short delays.

INDUSTRIAL

Industrial hardware and software. Functionality and security of the device has been assessed by important actors of the building automation and IoT field.

FLEXIBLE

The advanced routing feature permits to support all use cases. Conditional routing, value transformations, creation of JSON objects, alarms in case of communication issues, and more.

INTUITIVE

Visual and intuitive, the web interface makes the creation of simple routes easy with a drag & drop process.

EXPORT / IMPORT

All routes configuration can be easily exported and imported as a CSV or JSON file.

Universal Gateway

Building automation & IoT

Contents

1 ROUTES PARAMETERS 3
1.1 NAME 3
1.2 TYPE 3
1.3 SOURCE GATEWAY 4
1.4 SOURCE ADDRESS 4
1.5 SOURCE VALUE 5
1.6 DIRECTION 6
1.7 DESTINATION GATEWAY 6
1.8 DESTINATION ADDRESS 6
1.9 DESTINATION VALUE 7
1.10 REPEAT 7
1.11 DISABLE DELAY 7
1.12 MINIMUM DIFFERENCE 7
2 ADVANCED ROUTING EXAMPLES 8
2.1 REGULAR EXPRESSION ROUTE 8
2.2 STATUS ROUTE FOR ALARMS 9
2.2.1 BACNET OBJECT TO JSON MQTT 10

Universal Gateway – Routing

The routing table permits to transfer any type of serializable data (numerical values, strings, json objects, etc..) between all building automation and IoT protocols supported by the gateway software. Main standard open protocols are supported: KNX, Bacnet, Modbus, m-bus, MQTT, and more. The complete list of supported protocols can be found on our website at https://weble.ch. New protocols can be quickly integrated to the product on customer demand. The routing feature supports plain 1 to 1 routing between 2 addresses, but also more complex routing patterns detailed in this document. Due to the complex nature of universal/multi-protocol routing, advanced use-cases require basic JavaScript programming training.

1 Routes Parameters

Routes have several parameters described in this section. Main parameters are the source address, destination address, and direction of the routing. Source addresses listen to data-points change of values, and send those to the corresponding destination address. Routing can be performed from source to destination (default behaviour), reversed (destination to source), or bidirectional. Other parameters can be useful to transform the value before routing it, or to configure hysteresis to limit the traffic induced by frequent value updates.

1.1 Name

A name can be configured for each route. It is an optional parameter that is used as a description. It has no functional implications.

1.2 Type

There is 2 type of data/values that can be routed from a source address:

1. The source address **value**: usual value point. For Modbus for example it could be the integer value contained in a register.
2. The source address **status**: each address has also a status, indicating if its data-point value is available or not. The status is often used to trigger alarms if there is a communication issue. Main statuses are
    a. **Online**: means the source value is up to date and readable.
    b. **Offline**: means the driver process is not running, hence the source value is not up to date and readable.
    c. **Pending**: means the driver process is starting, so the address is not ready yet.
    d. Other: driver depending error messages. Custom status are implemented depending each communication protocol. For example, the **Modbus protocol** exposes Modbus specific error codes like **ILLEGAL_DATA_VALUE**, **ILLEGAL FUNCTION**, etc…

Based on the value, and status, there is 3 types of possible routes:

1. **Value:** Normal routes, the values are routed from one address to another. It is possible to change the direction of the routing.
2. **Status:** Status routes are routing a source address status, to a destination address value. This is useful to detect communication issues on an address, and trigger some events / alarms by writing a value to another protocol / device. As an example you could listen a Modbus status address, and if there is an error, send an alarm on Bacnet, or write a Bacnet object. Because the status is always read-only, the routing cannot be reversed or bidirectional: it is always routing a source status to a destination value. A detailed use case is described (Section 2.2).
3. **Value & Status:** In this case, both the source address value and source address status are triggering the route. Like for status routes, the direction of the routing cannot be changed. It is always routing from source to destination.

1.3 SOURCE GATEWAY

For each route, it is possible to precise the source gateway. If a device is running multiple gateways of the same protocol (e.g. 4 Modbus masters connecting to 4 different slaves), the system may end up with 4 occurrences of the same address name. In such cases the source gateway can be specified in the route (based on previous analogy, you can select one of the 4 Modbus masters available). (Note that the routes are not always 1-1 mappings. If the source gateway is left empty, all matching addresses trigger the route. For advanced use cases, instead of designing a single gateway, or all gateways, by leaving the field empty, it is also possible to use a regular expression or a JavaScript function to select a subset of matching gateways.)

1.4 SOURCE ADDRESS

The source address parameter defines the address that triggers the routing by listening new value changes on it. It can either be a single fixed address defined by its address name (simple case), or multiple addresses matching a pattern like a regular expression. All possible types of source address definition are listed below:

1. Single address **numerical identifier**. The address is identified with its internal id. This unique id is generated automatically by the system at address creation. Usually it is advised to use the address name instead.
2. Single address **name**. It is the usual way to design a single address. The address name is unique within its gateway, but if there is multiple gateways running the same protocol, it is possible to have colliding address names. To avoid this situation, the source gateway (section 1.3) can be specified.
3. Multiple addresses using a **regular expression (Regex)** pattern. This permits to have *n-1 and n-m* mappings. The route is triggered by all source addresses matching the regular expression (it lists all system addresses and test the regular expression against each address name). It permits to route multiple addresses with a single route. The computation of the destination address mapping is done using the pattern defined by the JavaScript string replace method (see use case at section 2.1).

4. JavaScript **function** pattern. With the online code editor, it is possible for programmers/advanced users to define a function that filters which addresses are triggering the route or not. Here is an example of a destination address function:

```
function(adrName, adrId, adr){
    //first parameter is the name of the address,
    //second parameter is the numerical id of the address,
    //third parameter is the full address object.
    //returns true to accept an address, or false to reject it.

    if(adrName.startsWith("%M300")){
        //accepts all modbus register starting with %M300
        //it could include %M300, %M3000, %M3001, %M30000, etc..
        return true;
    }else{
        //rejects the address: it will not trigger the route.
        return false;
    }
}
```

1.5 SOURCE VALUE

The source value field has 2 main use cases that can be combined together:

1. **Conditional routing** by filtering values that should not be routed.
2. **Transformation** of the value before writing it to the destination address.

If the field is left empty, then the original value is simply routed without alterations. When defined, it can take the following forms:

1. **Constant**: the original value is replaced (overwritten) by a constant fixed value. This constant can be binary, numerical, string, or anything except serializable, except JavaScript functions.
2. **Function**: in this case the source is a function that takes as parameter the routed value, process it, and returns the new transformed value. If the function returns nothing, the routing is aborted (conditional routing). Here is below a function that doesn't route values smaller or equal to 0, and multiple by 2 values greater than 0 (it means that a source value of 2 becomes 4 at the destination, and so on).

```
function(newValue, sourceAddress, destinationAddress, route){
    //first parameter is the new value being routed,
    //second parameter is the source address full object,
    //third parameter is the destination address full object,
    //last parameter is the route object itself.
    if(newValue > 0){
        return newValue * 2 //return the value times 2
    }else {
        return //returns nothings => abort routing.
    }
}
```

The source value is only used when the routing direction is from source to destination or bidirectional (see next section 1.6). Reversed routing is using the opposite parameter Destination Value (section 1.9).

1.6 DIRECTION

When the route *type* (section 1.2) is *value*, you can choose the route direction. The default direction is from source to destination. The following directions are possible:

Source > Destination. This is the default, **normal** routing.

Destination > Source. This is the **reversed** routing. The destination becomes the source, the source becomes the destination.

Source < > Destination. This is **bidirectional** routing. Values are routed in both ways. This routing mode, depending on the protocols / situations, can generate ping pong loops. To avoid the loops, the *Disable Delay* (section 1.11) and *Minimum Difference* (section 1.12) parameters are useful.

When the route *type* is *status* or *value/status*, the route direction is forced to **Source > Destination**. It is due to the address status being a read-only property.

1.7 DESTINATION GATEWAY

Same as *Source Gateway* (section 1.7), but applied to the destination address gateway. As an example, it is useful when routing Modbus register *%M100* of a given Modbus gateway "*modbusGwA"*, to Modbus register *%M100* of another Modbus gateway "*modbusGwB"*. In this scenario the source and destination addresses share the same name (*%M100*), so specifying which gateway is the source/destination is important. So the source gateway would be defined in the route as "*modbusGwA"*, and the destination gateway as "*modbusGwB"*.

1.8 DESTINATION ADDRESS

The destination address is the data-point to which the routed value is written/sent. It can either be a single fixed address defined by its address name (simple case), or multiple addresses matching a pattern. All possible types of destination address definition are listed below:

1. Single address **numerical identifier**. The address is identified with its internal id. This unique id is generated automatically by the system at address creation. Usually it is advised to use the address name instead.
2. Single address **name**. It is the usual way to design a single address. The address name is unique within its gateway. If there is multiple gateways running on the same protocol, it is possible to have colliding address names. To avoid this situation, the destination gateway (section 1.7) can be specified.
3. Multiple addresses using a **replace** pattern. This option needs to have the source address to be defined as a **regular expression**. The computation of the destination address mapping is done using the pattern defined by the JavaScript string replace method. (See use case section 2.1).
4. JavaScript **function** pattern. With the online code editor, it is possible for programmers to define a function that maps a given source address to a destination. The function returns the identifier of the destination address. The function use case is as below:

```
function(adrName, adrId, adr){
    //first parameter is the name of the source address,
    //second parameter is the numerical id of the  source address,
    //third parameter is the full source address object.
    //returns the destination address.
    if(adrName == '%M100'){
        return '%M200' //routes address %M100 to %M200
    }else{
        return '%M0' //routes all other addresses to %M0
    }
}
```

1.9 DESTINATION VALUE

The destination value is similar to the source value (section 1.5) concept, except it is used only when the route direction (section 1.6) is reversed or bidirectional. When routing from destination to source, this field can be used to transform the value, or abort the routing before writing the source address. When left empty, the value is routed to the source without transformations.

1.10 REPEAT

The repeat parameter permits to regularly (every *x* milliseconds) re-trigger / re-execute the route if the destination value does not equal to the source value. Sometimes the destination value can be inadvertently changed by a foreign entity/device/person. This parameters guarantees that the destination value matches the source value.

Note that if there is a communication failure on the destination gateway, the protocol driver process is restarted and retries to establish the connection. Once the gateway is connected, routes are automatically re-executed. It is **not** necessary to configure the repeat parameter for this use case.

1.11 DISABLE DELAY

Once a route is triggered and executed, it can be deactivated for a small duration like 100 milliseconds. It means that when the route is executed and the destination address written, potential new values coming from the source will be ignored during 100ms. This permits to limit excessive traffic and prevent routing loops that may occur

1.12 MINIMUM DIFFERENCE

The minimum diff parameter permits to define a hysteresis threshold for routing numerical values. If minimum difference is set to 0.1, then if the source value must vary by an increment superior to 0.1 to be routed to destination. Small value variations are hence ignored. This permits to limit unnecessary traffic.

2 ADVANCED ROUTING EXAMPLES

This section focuses on advanced routing examples based on real projects. (Simple routes can be easily configured using the web interface by drag & dropping an address on another).

2.1 REGULAR EXPRESSION ROUTE

This subsection demonstrates how to route multiple Bacnet objects to OPC UA with a single route. To achieve it, a regular expression will be used. Regular expression is a simple language/parser which permits to describe, capture, and replace patterns in any string of characters. In this use case it is used to identify matching source addresses, and transform each matching source address to a corresponding destination address. For a comprehensive description, please refer to the JavaScript implementation of regular expressions.

The following paragraph shows how to configure the route to map all Bacnet analog values properties to an OPC UA tree structure.

Route configuration:

1. **Source address**. By configuring the regex: `\%bac:local\/analogValue:(\d+)\/(.+)`
   - The above regex matches all Bacnet analog values properties. ○

   As an example, the following address names are matching:

   %bac:local/analogValue:1/presentValue
   %bac:local/analogValue:1/objectName
   %bac:local/analogValue:1/description
   %bac:local/analogValue:1/…
   %bac:local/analogValue:2/…
   %bac:local/analogValue:3/…
   …

   - A change of value in any of the matching addresses will trigger the route
   - The regex above include 2 capturing groups: `(\d+)\/(.+)`. The first one captures the Bacnet object instance number, and the second one the Bacnet property name.

2. **Destination address**. Replace pattern string
   - The prefix of the destination address corresponds to an OPC.
   - The $1 pattern refers to the first capturing of the source address regex.
   - As an example, the following bacnet addresses are mapped to opc addresses:
     %bac:local/analogValue:1/presentValue → %opc/analogValue/1/presentValue
     %bac:local/analogValue:1/objectName → %opc/analogValue/1/objectName
     %bac:local/analogValue:2/presentValue → %opc/analogValue/2/presentValue
     %bac:local/analogValue:*[instance]*/*[prop]* → %opc/analogValue/*[instance]*/ *[prop]*
   - Side note for programmers: each destination address is computed using the JavaScript JavaScript string replace method from the source address name, the regex, and the

replacement pattern as follows:

```
var regex = /\%bac:local\/analogValue:(\d+)\/(.+)/
var sourceAddress = "%bac:local/analogValue:1/presentValue"
var replacement = "%opc/analogValue/$1/$2"
var destinationAddress = sourceAddress.replace(regex,replacement)

//or in a single statement:
var destinationAddress = "%bac:local/analogValue:1/presentValue".replace(
                                    /\%bac:local\/analogValue:(\d+)\/(.+)/,
                                    "%opc/analogValue/$1/$2")

//the destination address name is "%opc/analogValue/1/presentValue"
```

In this example, note that the OPC UA destination addresses must exist: the route will not automatically create the missing OPC UA addresses. Addresses can be created either manually from the web interface, from a CSV or JSON file, or programmatically using the Weble open web API.

2.2 STATUS ROUTE FOR ALARMS

This section demonstrate how to create a route able to detect communication issues on a Modbus register, and if so trigger an email alert (SMTP driver).

By changing the route *type* (section 1.2), it is possible to not only route an address value, but also its status. The status is interesting to trigger alarms, watchdogs, when a given address is not responding or in faulty state. There exists 3 main status: “online”, “offline”, and “pending”. The Modbus driver also emits Modbus specific errors. These specific errors can be used to detect communication/configuration issues.

Let’s assume that the gateway is acting as a Modbus RTU master. It is gathering multiple values from different Modbus slaves. Each Modbus slave has a slave id. This id can be specified in the Modbus address: *%M1-10*0 represents the holding register number 100 of the slave with id 1. *%M2-100* represents the same register, but on the slave 2. If one of the slaves has a problem, those addresses will emit the error in their status. The error can either be a Modbus specific error, or a timeout error if the slave is not responding at all. The route can listen to changes of status, check if it is an error, and trigger an email alarm stating that the concerned Modbus slave is not responding.

Route configuration:

1. **Type.** The route type must be specified to **status**.
2. **Direction.** The direction is forced in source to destination (statuses cannot be written by routes).
3. **Source address.** For simplicity, a single ***%M1-100*** Modbus register address (Modbus slave 1) is used. Note that a regex could be used to have a single route covering all the slave statuses.
4. **Destination address.** A single email address (SMTP driver) ehelfer@weble.ch is configured.

5. **Source value.** The function returns the email text to be sent when the status indicates an error. If the status is normal, nothing is returned by the function and no email is sent.

```
function(status, sourceAddress){
    if(status == 'online' || status == 'pending' || status == 'offline'){
        //those are normal statuses sent when the modbus master is connecting.
        return;
    }else{
        //there is a modbus communication error like ILLEGAL_DATA or another.
        return "Modbus commication problem with address " + sourceAddress.name + "\n"+
               "Error: " + status; //body of the email to send
    }
}
```

6. **DisableDelay.** In this case, it may be a good define a disable delay of 20 minutes. If the address status is changing rapidly, it prevents sending too many emails alarms.

2.2.1 BACNET OBJECT TO JSON MQTT

The following example will demonstrate how to route a full Bacnet object with some of its properties into MQTT JSON messages. MQTT messaging protocol is lightweight and often used in IoT applications, like Amazon IoT cloud, Google IoT cloud, and other companies services.

The route will convert a Bacnet analog-value object present-value, object-name, and units, to the following JSON format:

```
{
    "properties": {
        "objectName": "BTU1 DELTA TEMP",
        "unit": "degreesCelsius"
    },
    "value": 3,
    "timestamp": "2019-10-28 18:49:39"
}
```

Note that a timestamp is added to the JSON. It corresponds to the date & time of the routing (when the analog-value object present-value changes and triggers the route).

Route configuration:

1. **Source address.** ***%bac:local/analogValue:1*** is the source address. It is a single local Bacnet object (the bacnet device is the weble gateway itself) of type analog-value. The value emitted on this address is the bacnet present-value.
2. **Destination address.** ***%mqtt:analogValue_1*** is the single destination MQTT topic. The address datatype must be configured as “json” for serialization.
3. **Source value.** The transform function is triggered whenever the Bacnet object present-value changes. The object name and units are stored in sub-addresses: *%bac:local/analogValue:1/objectName, %bac:local/analogValue:1/units*. These properties can be accessed in the route using our gateways API: https://api.weble.ch/module-core_gateways.html. The source value transform function is bound to the API (it means that the JavaScript “*this*” keyword refers to the API).

```
function bacnetToJson(value,sourceAddress,destinationAddress){
    //value here is the bacnet object present-value.
    var gwId = sourceAddress.gateway_id //get the bacnet gateway id.
    var boa = sourceAddress.name //e.g. %bac:local/analogValue:1
    // get the bacnet object properties, those are stored in sub-addresses.
    // other addresses can be obtained using the gateways api library (this).
    // https://api.weble.ch/module-core_gateways.html

    var objectName = this.getAddress({gateway_id: gwId, name : boa+'/objectName'}).value
    var objectUnit = this.getAddress({gateway_id: gwId, name : boa+'/units'}).value

    // returns the object, it will be serialized to json when written to MQTT.
    // The MQTT destination address must have the datatype "json" configured.
    return {
        "properties" : {
            "objectName" : objectName,
            "unit" : objectUnit
        },
        "value" : value,
        "timestamp" : (new Date()).toLocaleString() //adds a timestamp.
    }
}
```