BLGW | driver development guide

Ikatu
2016-10-10
1 Introduction 6
   1.1 Glossary ......................................................... 6

2 Driver structure 8
   2.1 Representation of resources ................................. 9
   2.2 Specification section ........................................ 10
   2.3 Functionality section ........................................ 10

3 Driver specification 12
   3.1 Generic field .................................................. 12
      3.1.1 Generic field examples ................................. 13
   3.2 Generic field builders ....................................... 14
      3.2.1 stringArgument ........................................... 14
      3.2.2 stringArgumentRegEx .................................... 14
      3.2.3 stringArgumentMinMax ................................... 15
      3.2.4 roStringArgument ........................................ 15
      3.2.5 numericArgument ......................................... 15
      3.2.6 roNumericArgument ...................................... 16
      3.2.7 passwordArgument ....................................... 16
      3.2.8 boolArgument ............................................. 17
3.2.9 floatArgument .................................................. 17
3.2.10 floatArgumentMinMax .................................... 17
3.2.11 enumArgument ............................................... 18
3.2.12 temperatureArgument .................................... 18
3.3 resource_types .................................................... 18
  3.3.1 resource_type ............................................... 19
3.4 driver_label ..................................................... 22
3.5 driver_help ........................................................ 23
3.6 driver_channels .................................................. 23
  3.6.1 TCP constructor ............................................. 24
  3.6.2 RS232 constructor .......................................... 24
  3.6.3 CUSTOM constructor ....................................... 26
3.7 driver_load_system_help ....................................... 26
3.8 driver_load_file_help .......................................... 26
3.9 Specification example .......................................... 27

4 Driver functionality ............................................. 30
  4.1 process ........................................................ 30
  4.2 executeCommand ................................................ 33
  4.3 onResourceDelete ............................................. 34
  4.4 onResourceUpdate ............................................. 34
  4.5 onResourceAdd ............................................... 35
4.6 Tools ................................................................. 35
4.6.1 The driver table .............................................. 35
4.6.2 The channel table ........................................... 36
4.6.3 The CONST table ........................................... 38
4.6.4 fireEvent ....................................................... 38
4.6.5 setResourceState ........................................... 39
4.6.6 monitorEvent ................................................ 39
4.6.7 readResource ................................................ 39
4.6.8 readAllResources .......................................... 40
4.6.9 Log ............................................................... 40
4.6.10 utils ............................................................ 40
4.7 resource Lua instance .......................................... 41
4.8 resource state Lua instance .................................. 41
4.9 Advanced features ............................................ 42
4.9.1 monitorEvent generalization ............................ 43
4.9.2 fireEvent ....................................................... 43
4.9.3 setResourceState generalization ....................... 44
4.9.4 Load resources .............................................. 44
4.9.5 Driver for systems providing Rest API for integration 52

5 Standard Resource Types ........................................ 62
5.1 Glossary ........................................................ 62
5.2 Standard resource types ........................................ 62
  5.2.1 Motivation and background ................................. 62
  5.2.2 Defined SRTs ................................................. 63
  5.2.3 State, commands and events ............................... 64
  5.2.4 Mandatory functionality .................................... 64

5.3 Identification of a command or event .......................... 64

5.4 BUTTON type .................................................... 65

5.5 DIMMER type ................................................... 66

5.6 SHADE type ..................................................... 67

5.7 THERMOSTAT_1SP type ......................................... 68

5.8 THERMOSTAT_2SP type ......................................... 69

5.9 GPIO type ....................................................... 70

5.10 AV_RENDERER type ............................................. 70
Introduction

Beolink Gateway (BLGW) provides connectivity with third party home automation systems.

Each system is different in the way it represents hardware and software resources (such as dimmers, sensors, actuators, keypads, variables), and also in the way it represents activity on those resources (for example, a button press).

Moreover, the actual communications protocol for interacting with external controllers is specific to each system brand or model.

The software inside BLGW that provides support for each third party system is called a *driver*.

In general, a driver will:

- Implement the *communication protocol* for interacting with the (third party) system.
- Provide an abstraction of elements in the system, and present them as standardized *resources* in BLGW.
- Provide all necessary *configuration settings* for the user, such as network addresses or authentication.

Most drivers in BLGW are written in the Lua programming language.

This document specifies the Lua interface provided by BLGW for the development of drivers, and instructions on how to add new Lua drivers to BLGW.

1.1 | Glossary

**Lua** A programming language which is well suited for extending other software. [http://www.lua.org/manual/5.2/](http://www.lua.org/manual/5.2/)

**Lua script** A text file containing a Lua program.

**System** A third party system to be supported by BLGW.

**Driver** Software inside BLGW to communicate with a particular system.

**Channel** A channel is an abstraction of a connection to a system. The supported connections are TCP and RS232, and for connectionless protocols there is a special channel named CUSTOM.
Resource A physical or logic element on the system installation, that will be represented in BLGW. Examples are: buttons, dimmers, shades, switches.

Resource type A specification for a resource. A driver must define one resource type for each kind of resource it supports. Whenever possible, a resource type should be mapped to a standard resource type, extending it as needed.

Standard resource type (SRT) One of a set of predefined resource types (for example, a button, or a dimmer). Standard resource types can be displayed on BLGW user interfaces. Whenever possible, they should be taken as a template for new resource types. This compatibility between resources for different drivers allows for generic programming, where an action can apply to many resources regardless of the underlying third party system.

Command An action performed by BLGW on a resource.

Event An external action on a resource, as detected by BLGW.

Monitoring Registering all events in order to assist identifying resources on a system.
A Lua driver consists of a single Lua script with two main parts:

**Specification** Settings and resource definitions.

**Functionality** Implementation of the protocols for interacting with the external system.

The following is a minimum example of a Lua driver that supports buttons with *press* and *release* commands and events. The protocol is very simple, with commands of the form *P123* and *R123*, where 123 is the button address.

The specification section defines the driver label, help, connection channels, address format, and resource types.

```lua
driver_label = "Simple system for demo"
driver_help = "Simple system help"
driver_channels = {
    TCP(2001, "192.168.1.10", "Ethernet", "TCP channel help")
}

resource_types = {
    ["Simple button"] = {
        standardResourceType = "BUTTON",
        address = stringArgument("address", "0"),
        events = { PRESS = {}, RELEASE = {} },
        commands = { PRESS = {}, RELEASE = {} }
    }
}
```

The functionality has two main methods: *process* reads the channel for incoming notifications and fires a BLGW event, and *executeCommand* is called by BLGW to send commands to the system.

```lua
local function processMessage(msg)
    local command = msg:sub(1,1)
    if command == "P" or command == "R" then
        local address = msg:sub(2,#msg-2)
        if command == "P" then
            fireEvent( "PRESS", "Simple button", address )
        elseif command == "R" then
            fireEvent( "RELEASE", "Simple button", address )
        end
    end
end
```
```lua
function process()
    Trace("process starting")
    if channel.status() then
        driver.setOnline()
    end
    while channel.status() do
        local msgError, msg = channel.readUntil("\r\n")
        if msgError == CONST.OK then
            processMessage(msg)
        end
    end
    channel.retry("Connection failed, retrying in 10 seconds", 10)
    driver.setError()
    return CONST.HW_ERROR
end

function executeCommand(command, resource, commandArgs)
    local msg
    if "PRESS" == command then
        msg= "P" .. resource.address .. "\r\n"
    elseif "RELEASE" == command then
        msg= "R" .. resource.address .. "\r\n"
    end
    local err= channel.write(msg)
    if err ~= CONST.OK then
        Error("Failed to execute command")
    end
end
```

2.1 | Representation of resources

A resource is any addressable element on the external system, such as a button, status LED or dimmer.

BLGW defines *Standard Resource Types (SRT)* which are a set of top level resource types with a well defined minimum functionality plus some optional extra functionality.

For example, a standard button is identified by the name BUTTON, and must at least support a PRESS command. Extra functionality such as RELEASE and HOLD events and commands, PRESS event or button LED status, are optional.

Individual resources of the same type on a system are identified by an *address*, which is represented as a single printable string. This printable string can be viewed and edited by the user.
The driver must be able to map between the string representation of an address and the actual protocol messages.

For example, a button may be identified by a keypad address 5 plus a button number 2. The address may be of the form "5,2", but the driver will have to parse and generate protocol messages of the form "PRESS,KPD_05,BTN_02".

The specification section of the Lua driver will contain a structure to represent supported resource types with their address format.

### 2.2 Specification section

The mapping of BLGW resources to physical resources requires deciding which SRT to use for each type of resource, and a way of encoding all the data necessary to identify the resource into a single string representation (the address).

This is specified in the `resource_types` structure.

The driver must also define a `driver_label` which identifies the driver and is the name shown to the user, the `driver_help` with all the general help information for the driver (everything not fitting any inline help field), and the communications channels as a `driver_channels` structure.

The mentioned `resource_types` structure, `driver_channels` structure, `driver_label` and `driver_help` conform the `driver specification`.

The `driver specification` is loaded first so that BLGW can offer the driver to the user during setup.

The section `Driver specification` explains in detail each of the variables and structures needed for a driver.

### 2.3 Functionality section

The functionality of the driver is defined by a set of predefined functions.

When a command is executed on a resource, BLGW calls the `executeCommand` function.

When the active channel opens a connection to the third party system, it executes `process`. The `process` function has to establish a connection with the external system, and read all channel input to check for incoming events or state updates.
The driver can call a set of functions to tell BLGW of an incoming event or state update on a resource.

Also, to keep resource states synchronized with the actual physical resources, BLGW notifies the driver by executing `onResourceUpdate`, `onResourceAdd` or `onResourceDelete` when a resource is respectively updated, added or deleted during setup.

There is no need to return from `process` as long as the connection with the system is up. But for all other functions it is mandatory to return immediately.
3 | Driver specification

This section describes the structures that a driver must define.

The first subsection describes a structure named **Generic field**, which is used throughout the specification.

Following subsections describe each of the required variables; then the last subsection gives an example specification.

3.1 | Generic field

This section is for reference only, in order to provide a deeper understanding of how parameters are represented internally.

A generic field is a table that specifies a variable, including how to present it to the user and how to validate user input.

There is no need to work directly with this structure; **constructors** are given for each specific complex field.

The structure of a generic field is as follows:

- **name**: A string that identifies the field, for custom fields it must begin with underscore ("_").
- **type**: One of password, string, int, float, enum or temperature.
- **label**: A string to show on the UI.
- **default**: The default value of the field.
- **validation**: Table with validation data to be used by the UI:
  - **min**: Only for numeric types, provides lower limit to be used on validation.
  - **max**: Only for numeric types, provides upper limit to be used on validation.
  - **regex**: Only for string or password types, a valid regular expression to be used on validation.
  - **read_only**: Valid for any type, a boolean indicating the field cannot be modified (when true).
  - **hidden**: A boolean indicating the field is not visible in the UI.
- **disallow_empty**: A boolean indicating whether or not the field can be empty, defaults to *false* so if empty is a valid value it should be set to *true*.

- **transient**: A boolean indicating whether or not the field should be persisted or not, if not its value will be lost after a reboot and when saving and loading the configuration (for example when selecting a previous revision).

- **units**: Only for temperature type, indicates the temperature units for the field.

- **values**: List with the possible values for enum type.

- **context_help**: A string to show as context help on the UI.

### 3.1.1 Generic field examples

```lua
local someGenericFields= {
  myString= {
    name= "my string",
    type= "string",
    label= "my string label",
    default= "12",
    validation= {regex= "[0-9]*" },
    context_help= "Please insert a number"
  },
  myEnum= {
    name= "my enum",
    type= "enum",
    label= "my enum label",
    default= "8",
    values= { "first value", "8", "other one" },
    context_help= "Select one of them"
  }
}
```

When building a generic field with the following call to the *stringArgumentRegEx* constructor:

```lua
local myfield= stringArgumentRegEx("_thestring", "defaultval", ".*",
  { context_help="myhelp", hidden= true,
    read_only=true, transient= true,
    disallow_empty= true})
```

It is equivalent to:

```lua
local myfield= {
  name= ".thestring",
```
3.2 | Generic field builders

3.2.1 stringArgument

Returns a generic field for a string. It is defined as function stringArgument( name, default_value, optionalArgs) where:

• name: generic field name.

• default_value: generic field default.

• optionalArgs: Table with more data for the the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty, transient and read_only.

3.2.2 stringArgumentRegEx

Returns a generic field for a string with a regexp for validation. It is defined as function stringArgumentRegEx( name, default_value, re, optionalArgs) where:

• name: generic field name.

• default_value: generic field default.

• re: generic field validation.regex.

• optionalArgs: Table with more data for the the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty, transient and read_only.
3.2.3 stringArgumentMinMax

Returns a generic field for a string with given minimum and maximum length. It is defined as function stringArgumentMinMax(name, default_value, min_len, max_len, optionalArgs) where:

- name: generic field name.
- default_value: generic field default.
- min_len: Minimum acceptable length for the attribute.
- max_len: Maximum acceptable length for the attribute.
- optionalArgs: Table with more data for the the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty, transient and read_only.

3.2.4 roStringArgument

Returns a generic field for a read only string. It is defined as function roStringArgument(name, default_value, optionalArgs) where:

- name: generic field name.
- default_value: generic field default.
- optionalArgs: Table with more data for the the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty and transient.

Calling roStringArgument(name, default, {}) is the same as calling stringArgument(name, default, {read_only=true}).

3.2.5 numericArgument

Returns a generic field for a number with a given valid interval. It is defined as function numericArgument(name, default_value, min_val, max_val, optionalArgs) where:

- name: generic field name.
- default_value: generic field default.
• min_val: Minimum acceptable value for the attribute.

• max_val: Maximum acceptable value for the attribute.

• optionalArgs: Table with more data for the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty, transient and read_only.

3.2.6 roNumericArgument

Returns a generic field for a read only number. It is defined as function roNumericArgument(name, default_value, optionalArgs) where:

• name: generic field name.

• default_value: generic field default.

• optionalArgs: Table with more data for the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty and transient.

Calling roNumericArgument(name, default, {}) is the same as calling numericArgument(name, default, {read_only=true}).

3.2.7 passwordArgument

NOTE: remove this or add support to Model and the UI Returns a generic field for a password given minimum and maximum acceptable lengths; for future use as by now it is handled as a simple string (including the UI). It is defined as function passwordArgument(name, default_value, min_len, max_len, optionalArgs) where:

• name: generic field name.

• default_value: generic field default.

• min_len: Minimum acceptable length for the attribute.

• max_len: Maximum acceptable length for the attribute.

• optionalArgs: Table with more data for the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty, transient and read_only.
3.2.8 boolArgument

Returns a generic field for a boolean argument. It is defined as function boolArgument(name, default_value, optionalArgs) where:

- name: generic field name.
- default_value: generic field default.
- optionalArgs: Table with more data for the the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty, transient and read_only.

3.2.9 floatArgument

Returns a generic field for a float argument. It is defined as function floatArgument(name, default_value, optionalArgs) where:

- name: generic field name.
- default_value: generic field default.
- optionalArgs: Table with more data for the the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty, transient and read_only.

3.2.10 floatArgumentMinMax

Returns a generic field for a float argument with a given minimal and maximal values. It is defined as function floatArgumentMinMax(name, default_value, min_val, max_val, optionalArgs) where:

- name: generic field name.
- default_value: generic field default.
- min_val: Minimum acceptable value for the attribute.
- max_val: Maximum acceptable value for the attribute.
- optionalArgs: Table with more data for the the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty, transient and read_only.
3.2.11 enumArgument

Returns a *generic field* for an enumerated argument. It is defined as function `enumArgument` (name, vals, default_value, validation, optionalArgs) where:

- **name**: *generic field* name.
- **vals**: The list of valid values for the argument.
- **default_value**: *generic field* default.
- **validation**: *generic field* validation, for read_only.
- **optionalArgs**: Table with more data for the the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty, transient and read_only.

3.2.12 temperatureArgument

Returns a *generic field* for a temperature argument. It is defined as function `temperatureArgument` (name, units, default, optionalArgs) where:

- **name**: *generic field* name.
- **units**: Units for the temperature argument, can be "C" or "F" for Celsius or Fahrenheit respectively.
- **default**: *generic field* default.
- **optionalArgs**: Table with more data for the the generic field, could be not present or empty. Accepts context_help, hidden, disallow_empty, transient and read_only.

3.3 | resource_types

*resource_types* variable must be a structure containing all the resource types the *driver* intends to handle, each should follow an *SRT* and specify its *address, commands, events and states* according to the *SRT*.

As specified in the *SRT section* there could be cases in which a resource in the third party system has no equivalent in BLGW (no *SRT* maps naturally), that being the case a non standard resource type can be used. Also in some cases there are resources which map naturally to an *SRT* but need to define some behaviour not defined in the *SRT*, for example a button on a third party system
which can handle a "multi tap" event. For those cases it is possible to define a resource type matching an SRT with a non standard functionality, as long as it defines all mandatory fields for the SRT it can define non standard commands, events and states. Non standard names must begin with an underscore (_) or will be rejected by BLGW.

The Lua driver must define a global variable named `resource_types` as a table which keys are strings that globally identify the resource type and its corresponding values are the specification for the resource type. For example being `simpleButtonType` and `LEDButtonType` variables containing the `resource_type` for "Button" and "LED button" respectively:

```lua
resource_types= {
  Button = simpleButtonType,
  ["LED button"] = LEDButtonType
}
```

### 3.3.1 resource_type

The values of the `resource_types` table are tables containing the following fields:

- **standardResourceType**: The name of the SRT this resource type intends to extend (e.g. "BUTTON", see the SRT documentation). If begins with an underscore it will be treated as a non standard resource type, meaning among other things that those resources will not have an UI representation.

- **address**: generic field that specifies the address format for the resource type, a way of checking its validity (typically a regular expression) and contextual help. Its name must be `address` and its type `string`.

- **events**: Table of events of the resource type.

- **commands**: Table of commands of the resource type.

- **states**: Table of states of the resource type.

- **context_help**: Contextual help of the resource type.

The driver should be able to identify resources given their name (key on the `resource_types` table) and address. Also when an event occurs on the third party system which the driver intends to handle, the driver needs to be able to build the address of the corresponding resource to notice BLGW.

For example lets define `simpleButtonType` used in the example in `resource_types` assuming this kind of resource is addressed by two decimal digits.
Notice the use of `stringArgumentRegEx` in order to simplify the code, the same could be done directly as follows:

```plaintext
simpleButtonType = {
    standardResourceType = "BUTTON",
    address = stringArgumentRegEx("address", "00", "[0-9]",
        {context_help = "Two digits address, e.g. 01" }),
    events = simpleButtonEvents,
    commands = simpleButtonCommands,
    states = simpleButtonStates,
    context_help = "This is a Button."
}
```

### 3.3.1.1 commands

Commands define the things that can be done from BLGW on the third party system for the corresponding resource. Standard commands are used on the UI and macros, non standard ones are only used on macros.

For each resource type a table of commands must be provided in which keys are the command names according to the standard e.g. PRESS or a non standard name which must begin with an underscore e.g. "_MULTI TAP", and values are tables with the following data:

- **context_help**: Context help of the command.
- **arguments**: A table with the command arguments as **generic fields** according to the SRT specification for the command, non standard arguments name must begin with an underscore.

An example of commands for the resource type "Button" of the previous example could be:
A bit more complex example implementing a resource type for the SRT "DIMMER" to present a command containing an argument follows:

```
resource_types= {
    ["simple dimmer"]= {
        standardResourceType= "DIMMER",
        address= stringArgumentRegEx("address", "00", "[0-9]")
        commands= {
            SET= {
                context_help= "Set the dimmer level.",
                arguments= {
                    numericArgument("LEVEL", 0, 0, 100,
                    { context_help= "The level of the dimmer (percentage)" } )
                }
            },
        },
        states= simpleDimmerStates
    }
}
```

### 3.3.1.2 events
Events by opposition of commands define things that happen in the third party system for the corresponding resource and are intended to be handled in BLGW. The same way as commands, for each resource type a table of events must be provided in which keys are the event names according to the standard e.g. PRESS or a non standard name wich must begin with an underscore e.g. "_MULTI TAP", and values are tables with the following data:

- **context_help**: Context help of the event.
- **arguments**: A table with the event arguments as generic fields according to the SRT specification for the event, non standard arguments name must begin with an underscore.

An example of events for the resource type "Button" of the previous example could be:

```
simpleButtonEvents= {
    PRESS= { context_help= "Single button press". }
}
```

Events and commands respond to the very same structure so in some cases code can be reused as in the following example:
3.3.1.3 states  Resource types could define state variables, which if defined must be kept synchronized with the third party system at any time (specially for standard resource types as loosing sync could result in a bad end user experience). The states field on the resource type is a list of generic fields, each defining a state variable intended to be handled on the resource. States of a resource type must follow the SRT for the resource type, and non standard state variables must be named beginning with an underscore.

A typical example of resource with state is a dimmer with feedback as shown in the following example (extending the example presented for command with arguments):

```
resource_types= {
    ["simple dimmer"]= {
        standardResourceType= "DIMMER",
        address= stringArgumentRegEx("address", "00", "[0-9]",
            {context_help= "Two digits address, e.g. 01" }),
        commands= {
            SET= { context_help= "Set the dimmer level.",
                arguments= { numericArgument("LEVEL", 0, 0, 100,
                    { context_help= "The level of the dimmer (percentage)" } ) } }
        },
        states= { numericArgument("LEVEL", 0, 0, 100) }
    }
}
```

3.4 | driver_label

The driver_label is a simple string to identify the system within BLGW, it is shown to the user during setup, and must be descriptive of the supported third party systems.
3.5 | driver_help

The `driver_help` is a simple string in Markdown language with the complete help information to allow an installer to make it work only by reading it and the inline help included in the driver (the `context_help` field present in many structures in the specification).

For example:

```
driver_help= [[[My driver help ===============
This is the help for the driver in markdown language.

---------

1. For code examples use ‘local var= 1’
2. There is __bold__ and **bold**, _italic_ and *italic*.

## Also
* bullets
* without
* numbers

And for a [link to google](http://google.com)
]]
```

3.6 | driver_channels

The `driver_channels` is a list (a number indexed Lua table) containing the specification for all channels which could be used to connect to the third party system using the same protocol (the one used on the driver), or with minor differences.

Three types of channel can be defined: RS232, TCP and CUSTOM, but more than one channel can be defined for each type.

For example, Lutron Homeworks Interactive provides integration through RS232 and TCP using the same protocol but with a different end of line mark. So in order to provide correct integration, RS232 and TCP channels should be defined, but an extra TCP channel can be provided to support RS232 over Ethernet, which will use the RS232 line ending.

All channel types defined here will be available for the user to choose from, and the one selected can be retrieved during runtime (see Driver functionality) and is referred to as the active channel.
Each channel type is described using a complex opaque table which can be built using the following constructors:

### 3.6.1 TCP constructor

Constructs a TCP channel to be used on `driver_channels`. It is a function defined as

```latex
function TCP( port, ip, name, help, args)
```

Which returns a structure valid for `driver_channels` table where:

- `port` is the default value for the TCP port to use in the connection,
- `ip` is the default value for the IP or hostname to connect to.
- `name` is the name of the channel, which is also displayed on the UI.
- `args` is a list (a Lua table) of generic fields with arguments, mostly for user defined parameters (e.g. login or password), plus eventually `timeBetweenMessages`, which is a numeric argument to set a minimum time in microseconds between outgoing messages (commands), in case the third party system has any time constraints on processing messages. User defined argument names must begin with underscore (_.) or will be rejected.

For example:

```latex
TCP(23, "192.168.1.3", "Direct Ethernet connection", "Direct Ethernet connection for our system",{
    stringArgumentMinMax("_login", "admin", 1, 10,
        { context_help = "User name for the system (factory default is \rightarrow admin)" } ),
    passwordArgument("_password", "admin", 1, 15,
        { context_help = "Password for the system (factory default is \rightarrow admin)" } )
})
```

### 3.6.2 RS232 constructor

Constructs an RS232 channel to be used on `driver_channels`.

It is a function defined as:
function rs232(pname, ptype, plabel, help, args)

Where:

- **pname**: Name for the channel.
- **ptype**: Type of the channel, only "rs232" allowed.
- **plabel**: Label for the channel.
- **help**: Help for the channel.
- **args**: A list (a Lua table) of generic fields, mostly for user defined parameters (e.g. login or password), plus eventually `timeBetweenMessages`, which is a numeric argument to set a minimum time in microseconds between outgoing messages (commands), in case the third party system has any time constraints on processing messages. And also to change the default value for one of the default arguments (to do this just define them with the intended values). User defined argument names must begin with underscore (_), or will be rejected.

For example:

```
rs232( "RS232", "rs232", "RS232 channel", "Direct connection through RS232",
    stringArgumentMinMax("_login", "admin", 1, 10,
        { context_help = "User name for the system (factory default is admin)" })
    , passwordArgument("_password", "admin", 1, 15,
        { context_help = "Password for the system (factory default is admin)" }))
```

### 3.6.2.1 RS232 Default arguments

The default RS232 arguments are:

- **dataBits**: read only numeric argument for data bits, defaults to 8.
- **stopBits**: read only numeric argument stop bits, defaults to 1.
- **baudRate**: enum argument for baud rate, defined with the values \{9600, 19200, 38400, 57600, 115200\} and 9600 as default.
- **parity**: enum argument for parity, defined with the values "Odd", "Even" and "None", using "None" as default.
- **inputMode**: enum argument for input modes, defined with the values "Canonical" and "Raw", "Raw" as default.
- **fcMode**: enum argument for flow control, defined with the values "None", "Soft" and "Hard", "None" as default.
3.6.3 CUSTOM constructor

Constructs a custom channel to be used on `driver_channels`. It’s a function defined as

```lua
function CUSTOM(name, help, args)

Which returns a structure valid for `driver_channels` table where:

- `name` is the name of the channel, which identifies it and is displayed on the UI.
- `help` is the help to display on the UI for the channel.
- `args` is a list (a Lua table) of _generic fields_ with user defined arguments (e.g. login or password). User defined argument names must begin with underscore (_) or will be rejected.

This channel is intended to be used to integrate with systems based on a Rest API using _URL functions_ instead of channel based ones.

For example:

```lua
CUSTOM("my connection", "help about this connection",
    {stringArgument("_baseurl", "http://192.168.1.1/"))
```

3.7 | driver_load_system_help

The `driver_load_system_help` global variable, if defined, means the driver is capable of loading resources from the system. Also its value is used as inline help on the UI for the load resources from system functionality.

When a driver defines this varialbe, it must provide an implementation for the `requestResources` function.

3.8 | driver_load_file_help

The `driver_load_file_help` global variable, if defined, means the driver is capable of loading resources from a file. Also its value is used as inline help on the UI for the load resources from file functionality.
When a driver defines this variable, it must provide an implementation for the `parseResources` function.

### 3.9 Specification example

Following is a simple specification example with two resource types: Button and Led. Button as a standard "BUTTON" and Led as a "GPIO". The example provides a basic specification which could be used to manage buttons and LEDs on a Lutron Radio Ra 2 system.

```
   driver_label= "example label"
   driver_help= [['example driver
   ===============

   This driver supports communication with Lutron Radio RA2.

   Connection to a Radio RA2 system
   --------------------------------

   Communication with Radio RA2 is done via the Radio RA2 Main Repeater, which allows interaction with the system via 100 programmable virtual buttons (*phantom buttons*).

   Connection settings consist of: IP address of the Main Repeater (default: 192.168.1.50), login (default: lutron), password (default: integration) and telnet IP port (default: 23).

   Resources
   -----------

   The supported resource types are:

   + **Button**: a keypad or control unit button.
   + **LED**: a single LED used for status.

   Resource address format
   ------------------------

   Resource addresses use *Integration ID* which by default is a number, but can also be a user defined string; and a sub-address called *Component Number*.

   # Availability of events and commands

   Lutron supports a lot of different hardware models and combinations.

   Not all hardware setups support the whole set of events and commands.

   Check the Lutron documentation (*Lutron Integration Protocol*), or use the monitoring facilities in BLGW to verify that the hardware actually
```
supports a command or event type.

A typical example is the 'MULTI TAP' event, which is available on a limited combination of Lutron hardware.

Events
--------------
+ Button
  - **PRESS**
  - **RELEASE**
  - **HOLD**
  - **MULTI TAP**: Pressing on the button repeatedly
  - **HOLD RELEASE**: Releasing a button after a long press (HOLD)

Commands
--------------
+ Button
  - **PRESS**
  - **RELEASE**
  - **HOLD**
  - **MULTI TAP**: Pressing on the button repeatedly
  - **HOLD RELEASE**: Releasing a button after a long press (HOLD)

Resource State
--------------
- **STATE**: The state of the LED (0 means OFF and 1 ON)

---

local TCP_arguments = {
  stringArgumentMinMax(".login", "lutron", 1, 10,
  { context_help= "User name for the lutron system (factory default is lutron)"}),
  passwordArgument(".password", "integration", 1, 15,
  { context_help= "Password for the lutron system (factory default is integration)" })
}

local buttonActions = {
  PRESS= {},
  RELEASE= {},
  HOLD= {},
  "MULTI TAP"= { context_help= "Pressing and releasing the button multiple times" },
  "HOLD RELEASE"= { context_help= "Releasing a button after a long press (HOLD)" }
}

-- positive number without zeros on the left
local positiveNumber= "\\([0-9]\|\[1-9][0-9]*\)\"
{ context_help= "Integration ID, Component Number (e.g. \"1,1\" or \"myIID,28\")" }

resource_types= {
  Button= {
    standardResourceType= "BUTTON",
    address= theAddress,
    events= buttonActions,
    commands= buttonActions
  },
  Led= {
    standardResourceType= "GPIO",
    address= theAddress,
    states= { enumArgument( "STATE", {0,1}, 0 ) }
  }
}
4 | Driver functionality

Once a specification is provided, the driver must implement some functions in order to interact with the third party system. Commands executed on the BLGW will result in calls to `executeCommand` function, which the driver must implement. The driver also needs to implement the `process` function to connect to the third party system using the active channel, detect events and notify BLGW. Also `onResourceAdd`, `onResourceDelete` and `onResourceUpdate` should be implemented to take action when a resource is added, removed or updated during BLGW setup.

4.1 | process

This is the main function that provides functionality to the driver. It is called whenever an active channel is selected an its corresponding port is succesfully opened.

It should use the provided API to:

1. Setup a connection to the driver, handling authentication if needed. On success, set the driver connection state accordingly.
2. Request all the handled resource states to the third party system.
3. While the active channel is connected, wait for incoming messages. When a message arrives corresponding to events or updates on state variables, report to BLGW through `fireEvent`, `setResourceState` or `monitorEvent`.

If the connection fails for any reason, the `process` function should return providing the value of `CONST` which best fits the case. It will be called again immediately or after some seconds depending on the returned value, unless the driver previously called `channel.retry` in which case the call will be delayed the requested time.

Follows a commented `process` function for the example specification given above.

1. Wait for a message indicating the third party system is expecting the login name and return after asking the system to retry in 10 seconds if the message does not arrive.

```plaintext
local res = channel.waitFor("login: ",1)
if res ~= CONST.OK then
  channel.retry( "Error while connecting to driver example system, it is not asking for login name; please check you are using the correct IP address.", 10 )
```

2. Send the login name, wait until the system asks for password and send it.

```lua
local messageToSend= channel.attributes("_login\r\n")
local ret= channel.write(messageToSend)
if ret ~= CONST.OK then
    return CONST.INVALID_CREDENTIALS
end
ret= channel.waitFor("password:",1);
if ret ~= CONST.OK then
    channel.retry( "Error while connecting to QS, system is not asking " .. "for password, please check you are using the correct IP address.", 10 )
    return CONST.TIMEOUT
end
messageToSend= channel.attributes("_password\r\n")
ret= channel.write(messageToSend)
if ret ~= CONST.OK then
    return CONST.INVALID_CREDENTIALS
end
```

3. Set the driver connection state to online and send some initialization messages (in this example, setting up the Lutron system to report everything that happens).

```lua
driver.setOnline()
channel.write("#MONITORING,255,1\r\n") -- all on but prompt and reply state
channel.write("#MONITORING,11,1\r\n") -- reply state on
channel.write("#MONITORING,1,2\r\n") -- diagnostic off
channel.write("#MONITORING,12,2\r\n") -- prompt off
```

Second step: request state of all resources (all resources with defined states, in our case only LEDs):

```lua
for res in readAllResources("Led") do
    getState(res)
end
```

Where `getState` is defined as follows:

```lua
local function getState(resource)
    if resource.typeId == "Led" then
        local iid, cn, ledcn= split(",", resource.address)
```
Finally the third step: While the connection is up and with the help of Lua string functions, we parse the received messages and call BLGW when corresponds.

```lua
local msg = "?DEVICE," .. iid .. ".. cn .. ",9\r\n"
Debug("getState: sending: " .. msg)
local err= channel.write(msg)
if err ~= CONST.OK then
    Error("getLedState: error trying to Get led state")
end
end

while channel.status() do
    local err, msg = CONST.OK, ""
    repeat
        err, msg = channel.readUntil("\r\n", 10)
        if err == CONST.TIMEOUT then
            channel.write("\r\n") -- keepalive
        end
    until err ~= CONST.TIMEOUT

    if err ~= CONST.OK or msg == "" then
        if err ~= CONST.OK then
            Trace("readUntil failed")
        else
            Trace("received empty message")
        end
        if err ~= CONST.TIMEOUT then
            return CONST.HW_ERROR
        else
            return CONST.OK
        end
    end

    local parsedCmd, iid, p1, p2, p3, p4, p5, p6= split("", msg)
parsedCmd= string.gmatch( parsedCmd, ".*~(%a+)")()
if parsedCmd then
    Debug("parsed cmd: " .. parsedCmd )
parsedCmd= parsedCmd:upper()
end

if parsedCmd == "DEVICE" then
    local cn, cmd, ledState= p1, tonumber(p2), tonumber(p3)
    if cmd ~= nil then
        local address= unsplit("", iid, cn)
        if cmd == 3 or cmd == 4 or cmd == 5 or cmd == 6 or cmd == 32 then
            fireEvent(numToCommand(cmd), "Button", address)
        elseif cmd == 9 then -- LED
            if ledState and ledState ~= 0 then
```
Where `numToCommand` is defined as follows:

```lua
local function numToCommand(cmd)
    if cmd == 3 then -- PRESS
        return "PRESS"
    elseif cmd == 4 then -- RELEASE
        return "RELEASE"
    elseif cmd == 5 then -- HOLD
        return "HOLD"
    elseif cmd == 6 then -- MULTI TAP
        return "_MULTI TAP"
    elseif cmd == 32 then -- HOLD RELEASE
        return "_HOLD RELEASE"
    elseif cmd == 9 then -- LED
        return "_LED"
    end
end
```

4.2 | `executeCommand`

This function is executed whenever a macro or a user from the UI executes a command on a resource. The function prototype must be:

```lua
function executeCommand(command, resource, commandArgs)
```

Where:

- `command` is the name of the command being executed, e.g. "PRESS".
- `resource` is the `resource` on which the command is being executed.
- `commandArgs` is a table containing all the arguments of the command, keys in the table being argument names and their associated values the argument values.
Remember this function must return as soon as possible, commands taking too long to execute result in bad user experience and unexpected behaviour on macro programming. On our example, the executeCommand implementation would be:

```lua
function executeCommand(command, resource, commandArgs)
    Trace("Command executed: ", command)
    if commandNumbers[command] then
        local iid, cn, lcn = split(",", resource.address)
        local cmd= commandNumbers[command]
        if resource.typeId == "Button" then -- our only resource type with commands.
            local err= channel.write("#DEVICE," .. unsplit(",",iid, cn) .. "," .. cmd .. 

end
end
end

Where commandNumbers is defined as:

```lua
local commandNumbers= { PRESS = 3, RELEASE = 4, HOLD = 5,
                       "_MULTI TAP" = 6, "_HOLD RELEASE" = 32, SET= 1 }
```

4.3 | onResourceDelete

This function is called whenever a resource of the system is deleted. It must be defined as function onResourceDelete(resource) where resource is the resource that was deleted. In our example there is not much to be done so we only log a message:

```lua
function onResourceDelete(resource)
    Trace("Resource was deleted")
end
```

4.4 | onResourceUpdate

This function is called whenever a resource of the system is updated. It must be defined as function onResourceUpdate(resource) where resource is the resource that was updated. Following the example when a resource is updated we should request its state as follows:
function onResourceUpdate(resource)
    Trace("Resource was updated")
    getState(resource)
end

4.5 | onResourceAdd

This function is called whenever a resource of the system is added. It must be defined as function onResourceAdd(resource) where resource is the resource that was added. For the example we do the very same as in onResourceUpdate:

function onResourceAdd(resource)
    Trace("a resource was added")
    getState(resource)
end

4.6 | Tools

4.6.1 The driver table

Provides functions to set the driver connection state, which is shown to the user as colored icons.

4.6.1.1 driver.setOffline()  Sets the system connection state to Offline.

4.6.1.2 driver.setConnecting()  Sets the system connection state to Connecting.

4.6.1.3 driver.setConnected()  Sets the system connection state to Connected.

4.6.1.4 driver.setOnline()  Sets the system connection state to Online.

4.6.1.5 driver.setError()  Sets the system connection state to Error.
4.6.2 The channel table

Provides functions to communicate with the third party system through the active channel.

4.6.2.1 channel.type() Retrieves the active channel type.

4.6.2.2 channel.waitFor() Opens the active channel port if not already open, and consumes the input until the given message arrives or a given timeout is reached. Is defined as function

\[
\text{waitFor(expectedString, timeout)}
\]

where:

- expectedString: The expected message.
- timeout: The timeout, 0 or nil means forever (\text{waitFor(expectedString)} is the same as \text{waitFor(expectedString, 0)}).

It returns CONST.OK if the expected message arrives within the timeout, CONST.TIMEOUT if timed out and CONST.PORT_CLOSED if the connection failed. Even if a timeout argument was not given or 0, the function may return a timeout.

4.6.2.3 channel.retry() Logs a message and saves a timeout so the next time process returns (intended for an immediate return) it will sleep the given timeout before calling again. The process function may be called sooner, e.g. if the active channel settings change. It is defined as function

\[
\text{retry(logMessage, timeout)}
\]

where:

- logMessage: The message to log.
- timeout: The time to sleep before calling process again.

**process function must return immediately after calling** channel.retry. If process function returns without calling channel.retry it will be called immediately in case it returned CONST.OK, or after up to a minute for other return values; also a predefined message will be logged informing the error and the timeout before the retry.

4.6.2.4 channel.write() Appends a message to the send queue; it will be dispatched immediately or not depending on the argument timeBetweenMessages of the active channel (see driver_channels). It is defined as function

\[
\text{write(message)}
\]
• message: The message to send.

Its return value can be safely ignored as it returns always `CONST.OK` (for backward compatibility reasons).

### 4.6.2.5 `channel.status()`

Returns `true` if the active channel is connected, `false` otherwise.

### 4.6.2.6 `channel.readUntil()`

Stores all input from the active channel until it reads a given string or reaches a certain timeout. If the given string arrives, it returns all the read data including the ending string. It is defined as function `readUntil(s, timeout)` where

- `s`: The string to wait for.
- `timeout`: The timeout, if not present or `0` it is set to infinity.

Returns two values `<r_1, r_2>` where `r_1` is the return code and `r_2` the message read (if `r_1` is `CONST.OK`, `r_1` is `CONST.OK` if read OK, `CONST.TIMEOUT` if a timeout was reached and `CONST.PORT_CLOSED` otherwise. Even when a timeout is not given, `CONST.TIMEOUT` may be returned.

### 4.6.2.7 `channel.attributes()`

Retrieves an attribute of the active channel. It is defined as function `attributes(name)` where:

- `name`: The attribute name.

And returns the value of the attribute.

### 4.6.2.8 `channel.writeHex()`

Appends a message given as a list of bytes to the send queue; it will be dispatched immediately or not depending on the argument `timeBetweenMessages` of the active channel (see `driver_channels`). It is defined as function `writeHex(hexMsg)` where:

- `hexMsg`: Message to send as a list of bytes.

Its return value can be safely ignored as it returns always `CONST.OK` (for backward compatibility reasons).
4.6.2.9  channel.setAttribute()  Sets a single attribute of the active channel given its name. It is defined as function setAttribute(name, value) where:

- **name**: The attribute name.
- **value**: The value to set the attribute to.

4.6.3  The CONST table

Constant values to return from process and returned by some functions:

- **OK**: Everything is OK.
- **HW_ERROR**: Unspecified error on the third party system/channel.
- **INVALID_CREDENTIALS**: Failed authentication in the third party system.
- **TIMEOUT**: Connection timed out.
- **PORT_CLOSED**: Connection port was closed.
- **CONNECTED**: Connection OK.
- **POLLING**: Indicates the caller to call again (used on process for rest polling).

4.6.4  fireEvent

Fires an event on a specific resource. If resource does not exist, a generic monitor event is generated so that it can be captured and assigned to a new resource. It is defined as function fireEvent(event, resourceType, address) where:

- **event**: Event name, e.g. "PRESS".
- **resourceType**: Resource type (key in resource_types).
- **address**: The resource address string, e.g. "1,2".

For example to fire a "PRESS" event on a resource type named "simple button":

```javascript
fireEvent("PRESS", "simple button", "1")
```
4.6.5 setResourceState

Sets a resource state or generates a corresponding monitor event if the resource does not exist. It is defined as function `setResourceState(resourceType, address, arguments)` where:

- `resourceType`: Resource type (key in `resource_types`).
- `address`: The resource address, e.g. "0:1:3".
- `arguments`: Key-value list for arguments specified in `resource_types[resourceType].states` (see `resource_types`).

If it finds a resource matching `resourceType` and `address`, it updates the corresponding state. In other case it generates a monitor event containing all the information. Updating the state of a resource to its same current state will not result in a state update event in BLGW. For example, for a resource type named "LED button" which accepts numbers as addresses, a call to update its state variable "STATE" to 1 should be:

```javascript
setResourceState("LED button", "1", { STATE = 1 })
```

4.6.6 monitorEvent

Generates a monitor event. It is defined as function `monitorEvent(message, resourceType, arguments)` where:

- `resourceType` is the resource type.
- `arguments` is a table containing data to attach to the monitor event, in most cases only the resource address.

4.6.7 readResource

Returns a resource given its type and address. It is defined as function `readResource(type, addr)` where:

- `type`: The resource type.
- `addr`: The resource address string.
4.6.8 readAllResources

Lua iterator over all the resources of a given resource type. It is defined as function `readAllResources(type)` where `type` is the resource type. It returns a `resource` on each iteration. For example, to iterate over all resources of type "my type" and call some function on each:

```lua
for resource in readAllResources("my type") do
    doSomething(resource)
end
```

4.6.9 Log

There are functions to log messages to the system log. All of them accept a message to log and optionally a boolean to indicate whether the message should be shown to the end user or not (defaults to true if not present). Each one is named after the log level it generates as follows:

- `function Fatal(message, user)` Fatal level log message.
- `function Error(message, user)` Error level log message.
- `function Warn(message, user)` Warning level log message.
- `function Info(message, user)` Info level log message.
- `function Debug(message, user)` Debug level log message.
- `function Trace(message, user)` Trace level log message.

4.6.10 utils

4.6.10.1 tableConcat Helper function to concatenate tables, defined as:

```lua
function tableConcat(t_1, ..., t_n)
```

Where:

- `t_i`: Is the table to concatenate in the position `i`.

Returns the table resulting of the concatenation of all the `t_i` sequentially.
4.6.10.2 map  Returns a table containing the elements of a given table, evaluated through a given function. It is defined as \texttt{function map \( f, \text{values} \)} where:

- \( f \): The function to map.
- \( \text{values} \): The table which elements are intended to be evaluated.

4.6.10.3 listMap  Receives a function and a list of elements and returns the list of elements evaluated through the function. It is defined as:

\begin{verbatim}
function listMap( f, e_1, ..., e_n)
\end{verbatim}

Where:

- \( f \): the function to map.

Returns all the received elements evaluated through the function \( f \).

4.7 | resource Lua instance

- Table with the following attributes:
  - typeId: Resource type label (see resource_type).
  - ID: Resource numeric id as a string.
  - name: Resource name (see resource_type).
  - states: Resource states (see resource state Lua instance).
  - address: Resource address.

4.8 | resource state Lua instance

- Table with a (key,value) for each state variable of a resource (key is the name, value the state).
4.9 | Advanced features

Ideally defining a specification results in a perfect mapping between the third party system and BLGW resources, but it may not be always the case.

A common example that fails to map perfectly is one in which the third party system has two kinds of button, with and without LED.

And when the third party sends an event notifying a button was pressed, there is no way to tell if the pressed button has an LED or not. In other words, both types of buttons are addressed the same way in the third party system.

In this case, the best way is to define two resource types: a button with state, and a button without state. Then let the installer decide whether the button has an LED or not by choosing the correct type.

To facilitate this there is a way to monitor events for multiple resource types and an address corresponding to each resource type, so that when a button is pressed in our example system, a monitor event can be generated for both types. The installer, through the capture interface, should notice that a button was pressed and it is a "button" or a "button with LED".

To generate this special (multi type) monitor event, a monitorEvent generalization is provided.

Also, when inside process a message is received from the third party system notifying a press on a button on the example mentioned, the driver should check whether there exists a resource of both types and if not, generate the monitor event.

To avoid that repeated code the fireEvent does both, fires an event if the resource exists, and if not, it generates the corresponding monitor event.

Now for this to work in the case of multiple possible resource types and addresses, a generalization of fireEvent is provided.

For the analogous case in which a resource state update is intended instead of firing an event, a generalization of setResourceState is provided.

Some third party systems provide, as a way to mitigate the ambiguity of the event messages, a functionality to list all the defined resources on the system. This functionality could be available directly through the integration protocol or by an export of their programming tool. BeoLink Gateway provides Load resources in order for a driver to be able to take profit of this kind of functionality, by importing both kinds of export and using the information within the capture and listing the imported resources for "one click" add in the UI.
4.9.1 monitorEvent generalization

Generates a monitor event. It is defined as:

```markdown
function monitorEvent(message,
    resourceType_1, arguments_1,
    ...,
    resourceType_n, arguments_n)
```

Where:

- `resourceType_i` is the resource type of the ith resource.
- `arguments_i` is a table containing data to attach to the monitor event (in practice only the resource address) for the ith resource.

4.9.2 fireEvent

Fires an event on a resource or generates a corresponding monitor event if no resource is found which matches the given data. It is defined as:

```markdown
function fireEvent(event, resourceType_i, address_i)
```

Where:

- `event_i`: event name for the ith event, e.g. "PRESS".
- `resourceType_i`: resource type of the ith resource (key in `resource_types`).
- `address_i`: address of the ith resource a, e.g. "0:1:3".

If a resource matching `resourceType_i` and `address_i` is found for one of the given tuples, it fires the corresponding event. Otherwise, it generates a monitor event containing all the possible tuples.

Multiple events in the same call is intended for events which could be valid for more than one BLGW resource type. Until the type is determined by the installer (adding the correct resource) only one entry shows up on the capture, as opposed to calling `fireEvent` multiple times which results in multiple entries on the capture.
4.9.3  setResourceState generalization

Sets a resource state or generates a corresponding monitor event if the resource does not exist. It is defined as:

```plaintext
function setResourceState(resourceType_i, address_i)
```

Where:

- `resourceType_i`: Resource type of the ith resource (key in `resource_types`).
- `address_i`: The address for the ith resource, e.g. "0:1:3".
- `arguments_i`: Key,value list for arguments specified in `resource_types[resourceType].states` (see `resource_types`) for the ith resource.

If a resource matching `resourceType_i` and `address_i` is found for one of the given tuples, it updates the corresponding state. Otherwise it generates a monitor event containing all the possible tuples.

Updating the state of a resource to its current state will not result in a state update in BLGW (will not fire macros corresponding to a state update to that value nor be noticeable through system monitor).

Updating state of multiple resources in the same call is intended for use when a state update is received which could be valid for more than one BLGW resource type (same way as in `fireEvent`).

4.9.4  Load resources

In BLGW systems can provide a set of candidate resources called Loaded Resources as a means to store resource information. This data can be used later by the user to add a resource, or to improve the capture functionality in cases the protocol offers limited information in the events.

Loaded Resources provide information about resources either known to the system and directly retrieved by a request, or loaded from a file (e.g. an export from the system programming tool).

To add Loaded Resources the driver must define the `requestResources` function and/or the `parseResources` function. The `requestResources` function allows the driver to send a request to the system which response will arrive to the `process` function loop (and add loaded resources asynchronously), or to make a request to the system using a connection and/or protocol other than the one
used for the normal operation and process its response (and add loaded resources synchronously).
The `parseResources` function allows the system to process a file uploaded by the installer from
the UI and add loaded resources from it. In order for the driver to use the Loaded Resources
when generating events for the capture functionality two functions are provided: `readAllLoadedResources` and `readLoadedResource`. Loaded resources are represented in Lua by the `Loaded Resources structure`, and can be added as resources using the `addDiscoveredResource` function.

As mentioned before, loaded resources can be used in two ways.

1. When adding new resources directly from Loaded Resources.
2. When the driver generates events for the capture functionality.

The usage of Loaded resources when adding new resources directly from the structure consists of
a list in the resources section of the BLGW UI which displays the available information for each
Loaded Resource in a row. Alongside the information an extra column allows the user to add the
Loaded Resource as a BLGW resource in one of two ways:

1. In the selected Area/Zone.
2. In a specific Area/Zone inferred from the `areaName`/`zoneName` fields of the Loaded Resource
   in case a matching area and a matching zone are defined in BLGW.

Also the entire row gets disabled in case the Loaded Resource matches a defined BLGW resource.
The matching is by default done by comparing the resource address and type with the address and
type fields of the loaded resource, but can be defined by the driver by providing an implementation
for the `equals` function.

4.9.4.1 Loaded Resources structure  Loaded resources contain information of resources in
the third party system intended to help the installer user add BLGW resources, either by capture
or directly selecting from the list of loaded resources for the system.

The Loaded Resource structure is a Lua table containing the following string fields:

- `name`: The proposed BLGW resource name.
- `type`: The proposed type for the BLGW resource.
- `areaName`: The proposed area name for the BLGW resource.
- `zoneName`: The proposed zone name for the BLGW resource.
- `address`: The proposed address for the BLGW resource.
- **description**: A description to help the installer understand which resource in the third party system this loaded resource refers, other than the already included in the previous fields (should be empty if there is nothing to add).

When adding BLGW resources directly from the loaded resources list in the UI, the **name**, **type** and **address** are assigned to the created resource while the **areaName** and **zoneName** are optionally used. When in BLGW exists an area named exactly as the **areaName** field and it contains a zone named exactly as the **zoneName** field, the UI shows alongside the “add” button an “add in area/zone” one which results in adding the resource in the corresponding area and zone.

The **description** field can be used by the driver to give the installer additional information about the resource on the third party system other than the included in the other fields, such as for example a reference to its location (when area and zone are not enough), colour, etc.

### 4.9.4.2 Loaded Resources related functions

The first two functions presented here: **requestResources** and **parseResources**, may be implemented by the driver to provide loaded resources. Then the **addDiscoveredResource** function can be used within **requestResources** and **parseResources** to add loaded resources, or even within the **process** function when the driver adds loaded resources using the same connection/protocol used for normal operation (thus implementing **requestResources** only to make a request to the third party system).

#### 4.9.4.2.1 requestResources

When the third party system provides a way to retrieve information of all defined resources through its integration protocol, the driver can exploit the functionality by defining this function. This functionality is triggered through a button in the BLGW programming UI and to be available, the driver must provide an implementation for this function and define **driver_load_system_help** in its specification.

There are two ways of implementing this function:

1. **Asynchronous** When the driver is using a TCP or RS232 connection for communication with the third party system, and thus **process function** is listening messages from the third party to generate events on BLGW. If the third party system integration protocol provides a request through the same channel to list the defined resources, this function should only send the request as the **process function** is already waiting for messages from the system. In this case the function sends the request and returns, it is in the **process function** where the Loaded Resources are added to BLGW when it receives the answer to the request, by calling **addDiscoveredResource** function.

2. **Synchronous** Some systems provide a way to retrieve the defined resources directly from the controller but through a different protocol, thus using a different connection or way of connecting. This being the case as it does not interfere with the ongoing connection, the Loaded Resources can be added within this function. The common use case for this is when
the third party system provides an URL where to download a file containing the project resources through HTTP, not to be confused with the case of `parseResources` in which it’s the installer who uploads a file.

This function must be defined as:

```javascript
function requestResources()
```

And return two values, the first one is a Boolean indicating whether the operation succeeded or not. The second return value is the number of added loaded resources if the first one is true, or the error message if the first one is false.

### 4.9.4.2.2 parseResources

Most third party devices are programmed through an external desktop application in which a project is defined containing the resources and behaviour of the system, and then transferred to the device. These applications sometimes handle meta information not used in the device itself but useful for the programmer, thus it is not available through the device integration protocol but could help the programmer when defining BLGW resources. That being the case and if the application provides a way of exporting/saving the project to a file, the driver can implement this function to load that file manually through BLGW UI. When `driver_load_file_help` is defined in the driver specification the BLGW UI allows the installer to upload the file, and when uploaded, BLGW executes this function using the file name and the file content as arguments.

This function must be defined as:

```javascript
function parseResources(data, fileName)
```

The `data` argument contains the file content as a string, and the `fileName` argument is the name of the uploaded file.

The body of the function should parse the contents of `data` looking for resources and generate `loaded resources` for the system.

The function must return two values, the first one is a Boolean indicating whether the operation succeeded or not. The second return value is the number of added loaded resources if the first one is true, or the error message if the first one is false.
4.9.4.2.3 addDiscoveredResource  
This function is the way for the driver to add a Loaded Resource whether in parseResources, in requestResources or in process, and it is defined as:

```
function addDiscoveredResource(dr)
```

Where `dr` is a Loaded Resource.

4.9.4.2.4 readLoadedResource  
Allows the driver to request a single loaded resource given its address and type, it is defined as:

```
function readLoadedResource(resourceType, address)
```

Where `resourceType` is the type of the loaded resource and `address` its address. If there is a loaded resource for the system matching address and resource type, returns a Loaded Resources structure, otherwise returns nil.

4.9.4.2.5 readAllLoadedResources  
Allows the driver to request a list of all the loaded resources for a given type, it is defined as:

```
function readAllLoadedResources(resourceType)
```

Where `resourceType` is the type of the expected loaded resources. It returns a table which entries are Loaded Resources matching the given type.

4.9.4.2.6 equals  
On certain circumstances such as when displaying loaded resources on the UI to be added by the user, it is necessary to compare loaded resources with BLGW resources so the user can easily notice whether a loaded resource was added as BLGW resource or not. This is by default done by comparing address and type fields, a loaded resource is not presented to be added if there exists a BLGW resource with the same type and address.

This behaviour does not necessary fit any third party system, for instance on a system providing buttons, buttons with LED and LED’s, it might be expected for the driver to list given a button and a LED, three resources: the button, the LED and the button with LED.

Once the user adds a BLGW resource by selecting one of them, there are two loaded resources which should not show up any longer. If for example the user adds the button with LED, it makes no sense to keep displaying the LED and the button resources alone.
Being that the case, the comparison between loaded resources and BLGW resources would not fit as it would compare resource type, which is different, and address, which will also be different but, probably, the address of the LED and the address from the button could be inferred from the address of the button with LED. For that case this function is provided, a driver in need of a more complex comparison between loaded resources and resource types should provide an implementation of this.

The function must be defined as:

```lua
function equals(resource, loadedResource)

Where resource is a resource and loadedResource a Loaded Resource, and it must return true or false.

As an example, an implementation of the function to compare address and type and also name follows:

```lua
function equals(resource, loadedResource)
  return resource.address == loadedResource.address and
         resource.typeId == loadedResource.type and
         resource.name == loadedResource.name
end
```

4.9.4.3 Load resources examples

As an example lets suppose a third party system which controls buttons identifying them with unique numbers, organized by area and zone and having user defined names and a description. But, the provided protocol for integration only uses the numeric identifiers thus not providing names nor description or any other meta-data on the event messages. As buttons are uniquely identified by a number, this number is the natural address to use for BLGW resources. For the driver to be able to provide the friendly names and also zone and area names to BLGW installer, Loaded Resources could be used.

The third party system has three possible ways of providing the meta-data not present in the protocol which BLGW could exploit through the Lua driver, and here three examples are presented, one for each. Notice that the driver may use one, two or (more rarely) the three proposed solutions.

4.9.4.3.1 Example 1: File uploaded by the installer. Lets suppose the presented third party system offers a desktop application to define the buttons and their meta-data, and the application has an export tool which generates an CSV file containing a line for each defined button in the following form:
There is an intended obvious mapping to BLGW loaded resources for the sake of the example, on which we will use numeric id as address, name as name, area as areaName, zone as zoneName and description as description.

The driver should define `driver_load_file_help` and `parseResources` to process this file as follows:

```lua
function parseResources(data, fileName)
    local name, extension = split('.', fileName)
    if not extension or extension ~= '.csv' then
        return false, "Incorrect file."
    end
    local lines = splitToList("\n", data)
    for _, line in ipairs(lines) do
        local addr, n, area, zone, desc = split(',', line)
        addDiscoveredResource({
            type = "Our button",
            address = addr,
            name = n,
            areaName = area,
            zoneName = zone,
            description = desc
        })
    end
    return true, #lines
end
```

**4.9.4.3.2 Example 2: File retrieved directly from the third party system.** Now let’s suppose the third party system uses a tcp connection for integration, which the driver uses to send and receive events and commands, thus the driver has a tcp channel defined. Also, the third party system provides an CSV file exactly like the presented on the example 1 but in this case, it is not provided by a desktop app, but by the same system in the default HTTP port (80). That being the case the driver should define `driver_load_system_help` and `requestResources` to get the file and process it as follows:

```lua
function requestResources()
    local host = channel.attributes("host")
    local ret, msg = urlGet("http://" .. host .. "/TheFile.csv", ",", {})
    if ret ~= true then
        Error("Failed to request resources")
        return false, "Failed to request resources to system."
    end
end
```
Where `parseResources` is the one defined in the example 1.

### 4.9.4.3.3 Example 3: Loaded resources retrieved through integration protocol.

In this case let's suppose the same protocol used by the third party system to communicate events and commands, provides an operation to retrieve all defined resources and their meta-data. That being the case the driver should provide an implementation of `requestResources` which sends the request to the system, while its response is processed in the `process` function alongside the other messages from the third party system. Here we suppose that when the third party system receives a message with the string "GET_ALL_RESOURCES", it answers with several messages, the first one being "BEGIN_LIST_RESOURCES" and the last one "END_LIST_RESOURCES", while in the middle it sends messages in the form "RESOURCE,<ID>,<TYPE>,<NAME>,<AREA>,<ZONE>,<COMMENT>" for each defined resource. Similarly to the previous presented example here an implementation for the `requestResources` is provided and `driver_load_system_help` is defined, but now also an example implementation for `process` is provided as follows:

```
function requestResources()
  local err= channel.write("GET_ALL_RESOURCES\n")
  if err ~= CONST.OK then
    Error("Failed to request resources.")
    return false, "Failed to request resources to system."
  end
  return true, -1
end

function process()
  if ( not setup_connection() ) then
    channel.retry("Failed to initialize system, retrying in 10 seconds", 10)
    return CONST.HW_ERROR
  end
  driver.setOnline()
  local lrCount= 0
  while channel.status() do
    local err, msg= channel.readUntil("\r\n")
    if err ~= CONST.OK then
      channel.retry("Communication failed, retrying in 10 seconds", 10)
      return err
    end
    if isEvent(msg) then
      processEvent(msg)
    elseif msg == "BEGIN_LIST_RESOURCES" then
      -- process list of resources
    else
      -- process other messages
    end
  end
end
```

```
Info("Starting to add loaded resources")
elseif msg == "END_LIST_RESOURCES" then
  Info(tostring(lrCount) .. " resources were loaded")
lrCount= 0
else
  local cmd, id, rtype, rname, area, zone, comment= split(msg)
  if cmd == "RESOURCE" then
    local loadedResource= {
      address= id,
      type= sysTypeToBLGW(rtype),
      name= rname,
      areaName= area,
      zoneName= zone,
      description= comment
    }
    addDiscoveredResource(loadedResource)
lrCount= lrCount + 1
  end
end

4.9.5 Driver for systems providing Rest API for integration

Some systems provide a Rest API instead of a protocol over an established connection, for integration with such systems a CUSTOM channel should be used to prevent BLGW from trying to connect to the system and URL functions should be used instead of channel based ones. The main difference when working with a Rest API is within the process function on which instead of establishing a connection with the third party system and keeping it alive while receiving messages from the system, some kind of polling should be done. The way to implement a polling within process is by making all the needed requests at first, and then call channel.retry before returning CONST.POLLING telling BLGW to call process again after some seconds. For example:

... 

driver_channels= {
  CUSTOM("my connection", "connection help", { stringArgument("_baseurl", 
    "http://192.168.1.1/", {})})
}

... 

function process()
  local baseurl= channel.attributes("_baseurl")
  for res in readAllResources("My resource type") do
    local success, msg= urlGet(baseurl+res.address, ",", {}})
    if success then
      myUpdateResourceStateFromResponse(res, msg)
    end
  end
end
Some systems also provide a Rest streaming API to request state, in that case the stream URL functions should be used and the driver code should look much like non Rest drivers as process will be listening on a streaming request for state changes instead of polling.

4.9.5.1 URL functions

As a way to interact with HTTP servers easily without needing to implement the protocol directly over a TCP connection, functions are provided to solve the most common cases `urlGet` `urlPut` and `urlPost` to make an HTTP GET, HTTP PUT and HTTP POST request respectively.

4.9.5.1.1 urlGet

Performs an HTTP GET request to the given URL, it is defined as:

```lua
function urlGet(url, data, headers)

Where `url` is the URL to perform the GET operation, `data` is a string with the parameters, and `headers` is a table containing the corresponding headers for the request. `data` and `headers` are optional (as normally in Lua, if you want to set `headers` you must set `data`).

Returns two values, a flag indicating success and the response.

For example to perform an HTTP GET on myserver.com with argument `arg=28` on port 1892, setting ="User-Agent"= header to `BLGW`:

```lua
local ok, response= urlGet("http://myserver.com:1892/", "arg=28", { ["User-Agent"]= "BLGW" } )
if ok then
  processResponse(response)
end
```

Or to get from myserver.com on port 80:

```lua
local ok, response= urlGet("http://myserver.com")
if ok then
  processResponse(response)
end
```
To set headers but no data:

```lua
local ok, response = urlGet("http://myserver.com", ",", { ["User-Agent"]= "BLGW" })
if ok then
    processResponse(response)
end
```

4.9.5.1.2 urlPut  Performs an HTTP PUT request to the given URL, it is defined as:

```lua
function urlPut(url, data, headers)
```

Where `url` is the URL to perform the PUT operation, `data` is a string with the parameters, and `headers` is a table containing the corresponding headers for the request. Data and headers are optional (as normally in Lua, if you want to set headers you must set data).

Returns two values, a flag indicating success and the response.

4.9.5.1.3 urlPost  Performs an HTTP POST request to the given URL, it is defined as:

```lua
function urlPost(url, data, headers)
```

Where `url` is the URL to perform the POST operation, `data` is a string with the parameters, and `headers` is a table containing the corresponding headers for the request. Data and headers are optional (as normally in Lua, if you want to set headers you must set data).

Returns two values, a flag indicating success and the response.

4.9.5.2 Streaming URL functions  Streaming URL functions provide a way to interact with REST streaming APIs which some systems provide in order to receive real time updates, without polling. BLGW’s Rest streaming support is based on Firebase’s Streaming from the REST API, and being non standard it may not fit your system API. Nevertheless it should work given that the Rest streaming API consists on passing some kind of argument or header to an HTTP request resulting on that request never returning, but sending data as it is generated.
4.9.5.2.1  urlStreamCreate  Creates a streaming request for a given url, it is defined as:

```lua
function urlStreamCreate(request)
```

Where `request` is a table containing the following fields:

- `type`: the type of the request, must be "GET" or "POST".
- `url`: the URL to perform the operation.
- `headers`: a table containing the headers for the request.
- `arguments`: a string containing the parameters for the request.

Returns three values:

- A boolean flag indicating success.
- A table to identify the created stream on the other operations, it has an `id` field that uniquely identifies the stream but also the `url` and `type` fields of the `request` table which created it to allow easier identification.
- A string with information in case of error, `nil` in case of success.

Once a stream is created it can be used as argument for the following functions and when no longer needed it must be released calling `urlStreamDelete`.

4.9.5.2.2  urlStreamWait  Blocks the caller until there is data or an error on the given stream, it is defined as:

```lua
function urlStreamWait(timeout, stream_1, ..., stream_N )
```

Where `stream_1` to `stream_N` are the tables returned by successful calls to `urlStreamCreate`. `timeout` should be set to a number of seconds the function should wait for data before returning a timeout error (0 means infinity).

Returns three values:

- A boolean flag indicating success.
• A table containing the list of stream id’s that have news, and a function `has(stream)` which returns a boolean for a given stream indicating whether it has news or not.

• A table for error cases with the following fields:
  – type: can be `timeout` or `interrupt`, `timeout` means the timeout was reached and `interrupt` means `urlStreamInterrupt` was called.
  – `userdata`: only present when type is `interrupt`, a user defined table set in the call to `urlStreamInterrupt`.

### 4.9.5.2.3 urlStreamRead
Retrieves data from a given stream, it is defined as:

```lua
function urlStreamRead(stream)
```

Where `stream` is a table returned by a successful call to `urlStreamCreate`. Returns three values:

• A boolean flag indicating success.

• A table containing the results with the following fields:
  – url: the url of the request.
  – data: the returned data.
  – id: the stream identifier.
  – finalized: a boolean indicating whether or not the request ended.

• A string with a message in case of error.

### 4.9.5.2.4 urlStreamDelete
Releases the resources of a given stream returned by a successful call to `urlStreamCreate`, it is defined as:

```lua
function urlStreamDelete(stream)
```

Where `stream` is the stream to be deleted. Returns two values:

• A boolean indicating success.

• A string message in case of error.
4.9.5.2.5 urlStreamInterrupt  Interrupt an ongoing call to urlStreamWait, it is defined as:

```plaintext
function urlStreamInterrupt(userData)
```

Where `userData` is a user defined table to be returned in the interrupted call to `urlStreamWait` in case of success.

Returns two values:

- A boolean indicating success.
- A string message in case of error.

4.9.5.2.6 Rest streaming examples

1. Basic example For a service providing a boolean state according to the Firebase Rest streaming API at `http://www.somesite.com/state`, we build a non standard resource type to hold that state and keep it in sync using our rest streaming API.

```plaintext
driver_channels = {
  CUSTOM("custom channel", "Connection to the site.", {})
}
local url= "http://www.somesite.com/state"
local headers= {{["Accept"]= "text/event-stream"}}
resource_types = {
  ["Our state resource"] = {
    standardResourceType = "_OUR_CUSTOM_STATE",
    address = stringArgumentRegEx( "address", "", "",
      { context_help= "Force an address to forbid more than one resource." }),
    states = { boolArgument("_STATE", false, { context_help= "this is our state" }) },
    context_help = "Our state"
  }
}
...function process()
  local request= {}  request.arguments= ""
  request.url= url
  request.headers= headers
  request.type= "GET"
  local ok, stream, errmsg= urlStreamCreate(request)
  if ok then
```
28 | driver.setOnline()
29 | while ok do
30 |   local result, err
31 |   ok, result, err= urlStreamWait(10, stream)
32 |   if ok and result.has(stream) then
33 |     ok, result, errmsg= urlStreamRead(stream)
34 |     local data= string.gsub(result.data, "\n", "") -- throw away \n
35 |     if string.match(data, "event: put") then
36 |       local putData= tonumber(string.match(data, "event: put.*data: (.*)$"))
37 |       setResourceState("Our state resource", ",", { "_STATE" = putData })
38 |     end
39 |   end
40 | end
41 | end
42 | driver.setOffline()
43 | channel.retry("Something went wrong, retrying in 20 seconds", 20)
44 | return CONST.TIMEOUT
45 | end
46 | end
47 |...

2. Advanced example Now our previous example’s service provides ten different state values at http://www.somesite.com/N where N is 0 to 9 for each state respectively. So we change the url from:

```
local url= "http://www.somesite.com/state"
```

To:

```
local url= "http://www.somesite.com/
```

The address from:

```
stringArgumentRegEx("address", ",", ",", { context_help= "Force empty to allow only one." })
```

To:

```
stringArgumentRegEx("address", ",", "[0-9]", { context_help= "id of the resource." })
```

The `urlStreamCreate` from:

```
local request= {}
request.arguments= ""
request.url= url
request.headers= headers
request.type= "GET"
local ok, stream, errmsg= urlStreamCreate(request)
```

To:
for i = 0, 9 do
    local request = {
        request.arguments = "",
        request.url = url + tostring(i)
    }
    request.headers = headers
    request.type = "GET"
    local ok, stream, errmsg = urlStreamCreate(request)
end

And finally the setResourceState call from:
setResourceState("Our state resource", "", { "_STATE" = putData })

To:
setResourceState("Our state resource",
    result.url:sub(#result.url, #result.url),
    { "_STATE" = putData })

And it works like a charm, but lets say the user has to pay for each resource its state is requested, then we need to be able to only request the state of defined resources, so instead of creating streams for each number we only create one for each defined resource:

for resource in readAllResources("_OUR_CUSTOM_STATE") do
    local request = {
        request.arguments = "",
        request.url = url + tostring(i)
    }
    request.headers = headers
    request.type = "GET"
    local ok, stream, errmsg = urlStreamCreate(request)
end

Now we only do the needed requests, the capture won’t show the resources as there won’t be any resource state update for non defined resources, but that can be solved in some other way (e.g. see Loaded resources). The problem is that if the installer adds a new resource while online, its state won’t be synchronized until next time a rest streaming request fails. This is what urlStreamInterrupt is meant for, each time a resource is added or updated, we call urlStreamInterrupt and for the sake of simplicity at process we return to start over in that case:

driver_channels = {
    CUSTOM("custom channel", "Connection to the site.", {})
}
local url = "http://www.somesite.com/"
local headers= {"Accept"]= "text/event-stream"
resource_types = {
    ["Our state resource"] = {
        standardResourceType = "_OUR_CUSTOM_STATE",
        address= stringArgumentRegEx("address", "", "[0-9]", {context_help= "id of the resource."})
    }
}
states = { boolArgument("_STATE", false, { context_help= "this is our state" } ) },
context_help = "Our state"
}

function process()
  local myStreams= {}
  for resoruce in readAllResources("_OUR_CUSTOM_STATE") do
    local request= {}
    request.arguments= 
    request.url= url + tostring(i)
    request.headers= headers
    request.type= "GET"
    local ok, stream, errmsg= urlStreamCreate(request)
    if not ok then
      cleanup(myStreams)
      driver.setOffline()
      channel.retry("Failed to create stream, retrying in 20 seconds", 20)
      return CONST.TIMEOUT
    else
      table.insert(myStreams, stream)
    end
  end
  if ok then
    driver.setOnline()
    while ok do
      local result, err
      ok, result, err= urlStreamWait(10, stream)
      if ok then
        for _, stream in ipairs(myStreams) do
          if result.has(stream) then
            ok, result, errmsg= urlStreamRead(stream)
            local data= string.gsub(result.data, "\n", "") -- throw away \n
            if string.match(data, "event: put") then
              local putData= string.match(data, "event: put.*data: {"state":(.*)}$")
              putData= tonumber(putData)
              setResourceState("Our state resource",
              result.url:sub(#result.url, #result.url),
              { "_STATE" = putData })
            end
          end
        end
        else if err.type == "interrupt" then
          cleanup(myStreams)
          channel.retry("Resources changed, will start over in 5 seconds to keep it simple.", 5)
          return CONST.TIMEOUT
        end
      end
    end
    cleanup(myStreams)
    driver.setOffline()
channel.retry("Something went wrong or a resource was added, retrying in 20 seconds", 
  → 20)
  return CONST.TIMEOUT
end

function cleanup(myStreams)
  for _, stream in ipairs(myStreams) do
    urlStreamDelete(stream)
  end
end

...
5 | Standard Resource Types

5.1 | Glossary

**Resource** An abstraction inside BLGW architecture that represents a physical or logical entity, such as a button on a keypad.

**SRT** Standard Resource Type

**BLGW** BeoLink Gateway

**MLGW** Masterlink Gateway MkII

**MLGW Protocol** Control protocol to interact with MLGW.

**GPIO** General Purpose Input / Output, a signal with 2 logical levels (high/low, on/off, true/false, etc.).

**HIP** Home Integration Protocol

**Driver** Software inside BLGW or MLGW that supports a particular 3rd party system, including connection and interaction with that system, and transforming that interaction into the gateway’s internal representation.

5.2 | Standard resource types

5.2.1 Motivation and background

The specific functionality and features supported by 3rd party systems is very varied.

For example, something as simple as a button, may be considered differently on different systems:

- May support a simple PRESS event, or full PRESS, HOLD, RELEASE sequence, DUBLE/MULTI TAP, or LONG PRESS, or simply TRUE / FALSE state.
- May have an associated LED indicator, which you can query or even control.
- May be considered as part of a keypad, or as a completely independent entity.
- May just generate an event when pressed, or can also be commanded to emulate user activity.
MLGW drivers considered all the peculiarities of 3rd party systems, thus providing maximum flexibility for defining macros.

However, it was not possible to provide a common representation or functionality common to all buttons, and therefore the user interfaces (such as BeoLink App or the Web Panel) could not provide access to all resources, buttons included. Moreover, an abstraction called virtual button was defined to be able to call macros from the user interfaces or controllers via MLGW protocol.

BLGW introduces the concept of standard resource types. Each type provides a common functionality that should be supported by all drivers.

For example, a standard button provides PRESS, HOLD and RELEASE actions, independent of the 3rd party system features. Macros can now respond to any button in the system, and user interfaces can represent a button, without caring for the specifics of the actual hardware.

This minimum functionality can be extended by drivers in order to accommodate extra features.

5.2.2 Defined SRTs

Driver implementations must try to match all their resources into the defined SRTs.

A driver is allowed to define new types, or to extend the functionality of a SRT.

By convention, all SRTs and their standard actions are identified by a symbol. For extensions to SRTs, or for new types, the corresponding symbols start with an underscore. For example, the standard BUTTON and PRESS, or the non-standard _MULTI TAP.

The following are the SRTs already defined:

<table>
<thead>
<tr>
<th>SRT</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>BUTTON</td>
</tr>
<tr>
<td>Dimmer</td>
<td>DIMMER</td>
</tr>
<tr>
<td>Shade</td>
<td>SHADE</td>
</tr>
<tr>
<td>Thermostat 1 setpoint</td>
<td>THERMOSTAT_1SP</td>
</tr>
<tr>
<td>Thermostat 2 setpoints</td>
<td>THERMOSTAT_2SP</td>
</tr>
<tr>
<td>GPIO</td>
<td>GPIO</td>
</tr>
<tr>
<td>A/V renderer</td>
<td>AV_RENDERER</td>
</tr>
</tbody>
</table>
5.2.3 State, commands and events

Resources can keep one or more state attributes that can be queried at any time. For example, the current level of a light dimmer is a state attribute.

A command is an action that BLGW performs on the resource. For example, changing the channel on a TV.

An event is an indication that some activity has occurred on a resource. For example, a button has been pressed.

In many cases, events and commands coincide. Such is the case of a button press, which is both a command and an event.

Command and event interactions may contain attributes.

State changes in a resource generate a STATE_UPDATE event, which contains all state attributes.

5.2.4 Mandatory functionality

Each standard resource type will have a minimum mandatory functionality implemented by the driver.

This means that if the driver exposes a resource as one of the standard types defined, it is required to implement a minimum set of commands, events and state information.

Mandatory functions are marked with \((M)\) in the listings below.

5.3 Identification of a command or event

A resource is uniquely identified by the combination of area, zone, type and name, and is represented uniquely in string form as a path. For example:

Guest house/Kitchen/AV_RENDERER/BeoVision/

An event or command is represented by a resource path followed by an action (event or command), optionally followed by attributes and values.
Example of a simple command, and a command with 2 attributes:

Guest house/Kitchen/BUTTON/Lights ON/PRESS
Guest house/Kitchen/AV_RENDERER/BeoVision/Beo4 command?Command=TV&
   Destination selector=Video_source

Example state change event, with 1 attribute.

Guest house/Kitchen/BUTTON/Lights ON/STATE_UPDATE?STATE=1

Example generic event matching all state updates (see documentation for *generic programming*):

*//*//*/STATE_UPDATE

5.4 | BUTTON type

Commands and events:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESS (M)</td>
<td>evt / cmd</td>
<td>Button press.</td>
</tr>
<tr>
<td>HOLD</td>
<td>evt / cmd</td>
<td>Button being held pressed.</td>
</tr>
<tr>
<td>RELEASE</td>
<td>evt / cmd</td>
<td>Button released.</td>
</tr>
<tr>
<td>TAP</td>
<td>command</td>
<td>Button press and immediate release.</td>
</tr>
<tr>
<td>STATE_UPDATE</td>
<td>event</td>
<td>State update notification.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Used by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>STATE_UPDATE</td>
<td>Button feedback (LED).</td>
</tr>
</tbody>
</table>

The driver must ensure that button events follow the sequences:

- PRESS + RELEASE if the button is not being held
- PRESS + HOLD + RELEASE if the button is held. Only one HOLD event should be generated.
The state of a button is an integer value in the range 0 to 9 with 0 meaning OFF. 1 meaning ON. Optional values greater than 1 indicate other ON states and are driver dependent.

Example command:

Upstairs/Bedroom/BUTTON/Lights on/PRESS

Example events:

Social/Entrance/BUTTON/Vacation/RELEASE
Social/Entrance/BUTTON/Vacation/STATE_UPDATE?STATE=1

5.5 | DIMMER type

Commands and events:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET (M)</td>
<td>command</td>
<td>Set a dimming level.</td>
</tr>
<tr>
<td>SET COLOR</td>
<td>command</td>
<td>Set the color value for the dimmer.</td>
</tr>
<tr>
<td>STATE_UPDATE</td>
<td>event</td>
<td>State update notification.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Used by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEVEL</td>
<td>SET</td>
<td>Requested dimmer level.</td>
</tr>
<tr>
<td>COLOR</td>
<td>SET COLOR</td>
<td>Requested color value.</td>
</tr>
<tr>
<td>LEVEL</td>
<td>STATE_UPDATE</td>
<td>Dimmer level feedback.</td>
</tr>
<tr>
<td>COLOR</td>
<td>STATE_UPDATE</td>
<td>Color value feedback.</td>
</tr>
</tbody>
</table>

Dimming levels are an integer value from 0 (off) to 100 (fully on); this is valid both for the LEVEL state attribute and the SET command.

Color value is specified as a string containing the Hue, Saturation and Brightness values, in the format hsv(H,S,B)(https://en.wikipedia.org/wiki/HSL_and_HSV).

The H value is an integer from 0 to 360, and both S and B are an integer from 0 to 100.

Example commands:
Example events:

Upstairs/Bedroom/DIMMER/Downlight/STATE_UPDATE?LEVEL=32&COLOR=hsv(120,66,87)

5.6 | SHADE type

Commands and events:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAISE (M)</td>
<td>cmd</td>
<td>Shade starts raising.</td>
</tr>
<tr>
<td>LOWER (M)</td>
<td>cmd</td>
<td>Shade starts lowering.</td>
</tr>
<tr>
<td>STOP (M)</td>
<td>cmd</td>
<td>Shade stops.</td>
</tr>
<tr>
<td>RAISE</td>
<td>evt</td>
<td>Shade starts raising.</td>
</tr>
<tr>
<td>LOWER</td>
<td>evt</td>
<td>Shade starts lowering.</td>
</tr>
<tr>
<td>STOP</td>
<td>evt</td>
<td>Shade stops.</td>
</tr>
<tr>
<td>PRESET</td>
<td>command</td>
<td>Set preset level.</td>
</tr>
<tr>
<td>SET</td>
<td>command</td>
<td>Set specific level.</td>
</tr>
<tr>
<td>STATE_UPDATE</td>
<td>event</td>
<td>State update notification.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Used by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUM</td>
<td>PRESET</td>
<td>Requested shade preset.</td>
</tr>
<tr>
<td>LEVEL</td>
<td>SET</td>
<td>Requested shade level.</td>
</tr>
<tr>
<td>LEVEL</td>
<td>STATE_UPDATE</td>
<td>Dimmer level feedback.</td>
</tr>
</tbody>
</table>

The level parameter is an integer between 0 and 100. A level of 0 indicates a closed shade (minimum natural lighting), or lowered awning. The level 100 corresponds to an open shade, or raised awning (maximum lighting).

Preset numbers supported are 0 through 30.

Example command:

Upstairs/Bedroom/SHADE/*/PRESET?NUM=3
Example event:

Upstairs/Bedroom/SHADE/Left/STATE_UPDATE?LEVEL=45

5.7 | THERMOSTAT_1SP type

Commands and events:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET SETPOINT (M)</td>
<td>command</td>
<td>Set setpoint.</td>
</tr>
<tr>
<td>SET MODE (M)</td>
<td>command</td>
<td>Set operation mode.</td>
</tr>
<tr>
<td>SET FAN AUTO (M)</td>
<td>command</td>
<td>Set fan auto on/off.</td>
</tr>
<tr>
<td>STATE_UPDATE</td>
<td>event</td>
<td>State update notification.</td>
</tr>
<tr>
<td>SET SCHEDULE</td>
<td>command</td>
<td>Set schedule operation on/off.</td>
</tr>
<tr>
<td>SET ECO MODE</td>
<td>command</td>
<td>Set echo mode on/off.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Used by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>All SET commands</td>
<td>Value to set.</td>
</tr>
<tr>
<td>TEMPERATURE (M)</td>
<td>STATE_UPDATE</td>
<td>Local temperature readout.</td>
</tr>
<tr>
<td>SETPOINT (M)</td>
<td>STATE_UPDATE</td>
<td>Setpoint.</td>
</tr>
<tr>
<td>MODE (M)</td>
<td>STATE_UPDATE</td>
<td>Operation mode.</td>
</tr>
<tr>
<td>FAN AUTO (M)</td>
<td>STATE_UPDATE</td>
<td>Fan auto mode on/off.</td>
</tr>
<tr>
<td>SCHEDULE</td>
<td>STATE_UPDATE</td>
<td>Schedule operation on/off.</td>
</tr>
<tr>
<td>ECO_MODE</td>
<td>STATE_UPDATE</td>
<td>Eco mode on/off.</td>
</tr>
</tbody>
</table>

All on/off values are represented by true and false respectively.

Temperature settings are represented as decimal numbers (with optional decimal point).

Operation mode is one of:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>System off.</td>
</tr>
<tr>
<td>Heat</td>
<td>Heat only mode.</td>
</tr>
<tr>
<td>Cool</td>
<td>Cool only mode.</td>
</tr>
<tr>
<td>Auto</td>
<td>Auto heat/cool.</td>
</tr>
<tr>
<td>Em.Heat</td>
<td>Emergency heat mode.</td>
</tr>
</tbody>
</table>
5.8 | THERMOSTAT_2SP type

Commands and events:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET HEAT SP (M)</td>
<td>command</td>
<td>Set heat setpoint.</td>
</tr>
<tr>
<td>SET COOL SP (M)</td>
<td>command</td>
<td>Set cool setpoint.</td>
</tr>
<tr>
<td>SET MODE (M)</td>
<td>command</td>
<td>Set operation mode.</td>
</tr>
<tr>
<td>SET FAN AUTO (M)</td>
<td>command</td>
<td>Set fan auto on/off.</td>
</tr>
<tr>
<td>STATE_UPDATE</td>
<td>event</td>
<td>State update notification.</td>
</tr>
<tr>
<td>SET SCHEDULE</td>
<td>command</td>
<td>Set schedule operation on/off.</td>
</tr>
<tr>
<td>SET ECO MODE</td>
<td>command</td>
<td>Set echo mode on/off.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Used by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>All SET commands</td>
<td>Value to set.</td>
</tr>
<tr>
<td>TEMPERATURE (M)</td>
<td>STATE_UPDATE</td>
<td>Local temperature readout.</td>
</tr>
<tr>
<td>HEAT SP (M)</td>
<td>STATE_UPDATE</td>
<td>Heat setpoint.</td>
</tr>
<tr>
<td>COOL SP (M)</td>
<td>STATE_UPDATE</td>
<td>Cool setpoint.</td>
</tr>
<tr>
<td>MODE (M)</td>
<td>STATE_UPDATE</td>
<td>Operation mode.</td>
</tr>
<tr>
<td>FAN AUTO (M)</td>
<td>STATE_UPDATE</td>
<td>Fan auto mode on/off.</td>
</tr>
<tr>
<td>SCHEDULE</td>
<td>STATE_UPDATE</td>
<td>Schedule operation on/off.</td>
</tr>
<tr>
<td>ECO MODE</td>
<td>STATE_UPDATE</td>
<td>Eco mode on/off.</td>
</tr>
</tbody>
</table>

All on/off values are represented by true and false respectively.

Temperature settings are represented as decimal numbers (with optional decimal point).

Operation mode is one of:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>System off.</td>
</tr>
<tr>
<td>Heat</td>
<td>Heat only mode.</td>
</tr>
<tr>
<td>Cool</td>
<td>Cool only mode.</td>
</tr>
<tr>
<td>Auto</td>
<td>Auto heat/cool.</td>
</tr>
<tr>
<td>Em.Heat</td>
<td>Emergency heat mode.</td>
</tr>
</tbody>
</table>
5.9 | GPIO type

Commands and events:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET</td>
<td>command</td>
<td>Set value.</td>
</tr>
<tr>
<td>PULSE</td>
<td>command</td>
<td>Set on then off.</td>
</tr>
<tr>
<td>TOGGLE</td>
<td>command</td>
<td>Toggle current value.</td>
</tr>
<tr>
<td>STATE_UPDATE</td>
<td>event</td>
<td>State update notification.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Used by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>SET</td>
<td>Value of GPIO, true / false.</td>
</tr>
<tr>
<td>STATE</td>
<td>STATE_UPDATE</td>
<td>Value of GPIO, true / false.</td>
</tr>
</tbody>
</table>

Example command:

Garden/Pool/GPIO/Filter/SET?VALUE=true

5.10 | AV_RENDERER type

Commands and events:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beo4 command</td>
<td>command</td>
<td>IR simul low with default attributes.</td>
</tr>
<tr>
<td>Beo4 advanced command</td>
<td>command</td>
<td>Complete IR simul low.</td>
</tr>
<tr>
<td>All standby</td>
<td>cmd / evt</td>
<td>All products standby.</td>
</tr>
<tr>
<td>Light</td>
<td>event</td>
<td>Light function event.</td>
</tr>
<tr>
<td>Control</td>
<td>event</td>
<td>Control function event.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Used by</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Light and Control</td>
<td>Key action.</td>
</tr>
<tr>
<td>Command</td>
<td>Light, Control, B4, B4 adv</td>
<td>Key code.</td>
</tr>
<tr>
<td>Destination selector</td>
<td>B4 and B4 adv</td>
<td>Destination code.</td>
</tr>
<tr>
<td>Link</td>
<td>B4 adv</td>
<td>Link code.</td>
</tr>
<tr>
<td>Secondary source</td>
<td>B4 adv</td>
<td>Unit code.</td>
</tr>
</tbody>
</table>
Key actions can be one of Press, Continue or Key release.

Command key codes are one of:

- STANDBY, SLEEP, TV, RADIO, AUX_V_DTV2, AUX_A, VTR_V.MEM_DVD2, CDV_DVD, CAMCORDER_CAMERA, TEXT, V_SAT_DTV, PC, DOORCAM_V.AUX2, TP1_A.MEM, CD, PH_N.RADIO, TP2_N.MUSIC_USB, CD2_JOIN, VTR2_V.MEM2_DVD2, MEDIA, WEB, PHOTO, USB2, SERVER, NET, PICTURE_IN_PICTURE_P-AND-P
- CIFFER_0_Digit_0, CIFFER_1_Digit_1, CIFFER_2_Digit_2, CIFFER_3_Digit_3, CIFFER_4_Digit_4, CIFFER_5_Digit_5, CIFFER_6_Digit_6, CIFFER_7_Digit_7, CIFFER_8_Digit_8, CIFFER_9_Digit_9
- STEP_UP, STEP_DW, REWIND, REC_RETURN_RETURN, WIND, GO_PLAY, STOP, CNTL_WIND_Yellow, CNTL_REWIND_Green, CNTL_STEP_UP_Blue, CNTL_STEP_DW_Red
- MUTE, PICTURE_TOGGLE_P.MUTE, PICTURE_FORMAT_FORMAT, SOUND_SPEAKER, MENU, ANALOG_UP_1_Volume_UP, ANALOG_DW_1_Volume_DOWN, CINEMA_ON, CINEMA_OFF, OPEN_STAND_STAND
- CLEAR, STORE, RESET_INDEX, BACK, CMD_A_MOTS, GOTO_TRACK_LAMP, SHOW_CLOCK_CLOCK, EJECT, RECORD, MEDIUM_SELECT_SELECT, TURN_SOUND, EXIT
- CNTL_0_SHIFT-0_EDIT, CNTL_1_SHIFT-1_RAND, CNTL_2_SHIFT-2, CNTL_3_SHIFT-3_REP, CNTL_4_SHIFT-4_SEL, CNTL_5_SHIFT-5, CNTL_6_SHIFT-6, CNTL_7_SHIFT-7, CNTL_8_SHIFT-8, CNTL_9_SHIFT-9, C_REWIND_Continue_REWIND, C_WIND_Continue_WIND, C_STEP_UP_Continue_step_UP, C_STEP_DW_Continue_step_DOWN, CONTINUE_Continue_(other_keys), CNTL_C_REWIND_Continue_Green, CNTL_C_WIND_Continue_Yellow, CNTL_C_STEP_UP_Continue_Blue, CNTL_C_STEP_DW_Continue_Red, KEY_RELEASE
- FUNCTION_1, FUNCTION_2, to FUNCTION_40
- SELECT_Cursor_SELECT, CURSOR_UP, CURSOR_DW, CURSOR_LEFT, CURSOR_RIGHT

Destination code may be Audio_source, Video_source or V.TAPE/V.MEM.

Link code is one of Local_Default_source or Remote_source_(main_room).

Secondary source (unit) code may be one of DEFAULT or V.TAPE2_DVD2_V.MEM2.