# The supplementary commands for control-network communication protocol using in the SinBerBEST BIMG Test-Bedding

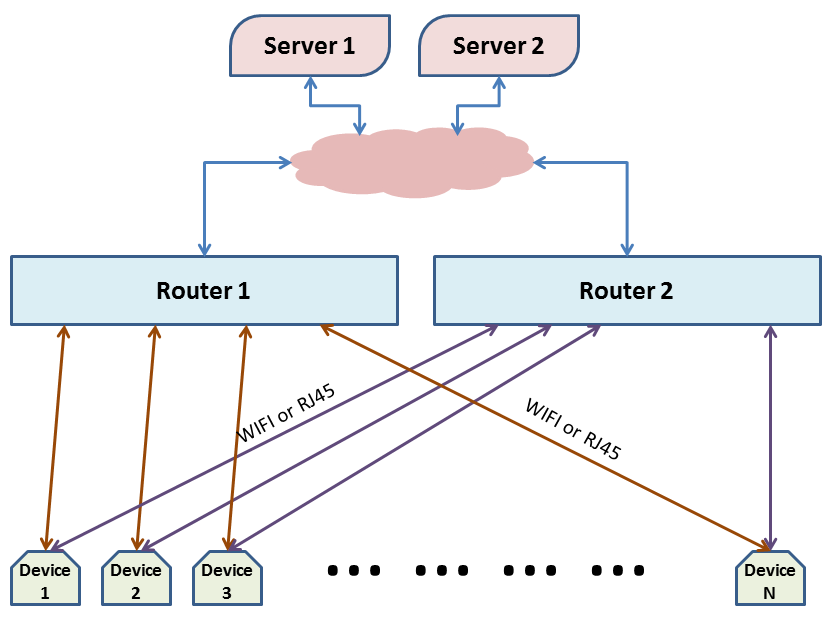
2014.09.24

This document presents the supplementary commands using in the SinBerBEST BIMG Test-Bedding, which is to enhance the control-network communication protocol based on websocket communication technique and multi-media LAN topology. It is focused on the commands during the initialization process and maintaining process of the control-network communication, such as connecting process and re-connecting process, in websocket protocol. The main purpose is to facilitate the SinBerBEST BIMG Test-Bedding’s control-network software development.

As the control commands are still undergoing the integration-test in the BIMG Test-Bedding dry lab, this document is only a draft and may be amended or extended later.

## 1. Background of the Control-Network design of the SinBerBEST BIMG Test-Bedding and its communication work principal

The control-network infrastructure of the SinBerBEST BIMG Test-Bedding is in a multi-media LAN topology (shown in Fig. 1). Supported by the infrastructure, the control-network can be established in a no-single-point-failure system.



WebSocket is a protocol providing [full-duplex](http://en.wikipedia.org/wiki/Full-duplex) communications channels over a single [TCP](http://en.wikipedia.org/wiki/Transmission_Control_Protocol) connection. It is designed to be implemented in [web browsers](http://en.wikipedia.org/wiki/Web_browser) and [web servers](http://en.wikipedia.org/wiki/Web_server), but it can be used by any client or server application. The WebSocket Protocol is an independent TCP-based protocol. Its only relationship to [HTTP](http://en.wikipedia.org/wiki/HTTP) is that its [handshake](http://en.wikipedia.org/wiki/Handshaking) is interpreted by HTTP servers as an [Upgrade request](http://en.wikipedia.org/wiki/HTTP/1.1_Upgrade_header). Normally, the handshake process is initialized by the client side. So it is a good practice that the devices are in automatic connecting and re-connecting websocket mode as they are located in the WiFi-router inside.

For convenience, the network IP of devices is setting to be DHCP. In this way, the network connecting path between a device and a server can be changed dynamically between (or among) routers in order to keep the communication in good condition.

## 2. Initialization and Maintenance Commands

When the websocket is established between a device and a server, the first thing to do is the exchange of the device name/server name as well as its Unique Identifier (UID). The highest byte of UID is reserved to be “0H” (i.e. only the lower 7 bytes is used for the UID). The highest byte of UID is to be used for the websocket Package-Data-Type (PDT). In initial commands (first connecting commands), for example, the PDT byte is setting to be “I” and to be “R” in re-connecting commands.

### The initial request/response commands used in communication connecting process:

The initial request command from device side in Java format is:

**ByteBuffer**[] requestInitialDevice = **new** **ByteBuffer**{long deviceUID, string deviceName};

The initial response command of server in Java format is:

**ByteBuffer**[] responseInitialServer = **new** **ByteBuffer**{long serverUID, string serverName};

The server is always in a listener mode. When a websocket is opened by a device-side connecting, the server expects the device’s initial request command. It won’t work until the initial request command is received. In fact, the service assignation to the device is based on the deviceName, and the device identification in the following websocket communication will use the deviceUID. The server returns the initial response command as soon as it received and identified the initial request command of device.

***note 1: to identify the device type, the definition of*** deviceName ***is recommended as follows:***

* ***"RTU-DC-Branch/yyy/xxx", where yyy is branch No., xxx (e.g.000, 001 or 004 ) is RTU type.***
* ***"RTU-AC-Branch/yyy/xxx", where yyy is branch No., xxx (e.g.002 ~ 004 ) is RTU type.***
* ***"RTU-AC/DC-BranchSwitch/yyy/xxx", where yyy is branch No., xxx (e.g. 004 ) is RTU type.***
* ***"RTU-DC/AC-CVT/yyy/xxx", where yyy is branch No., xxx (e.g.005 ) is RTU type.***
* ***\*"RTU-DC-Load/yyy/xxx", where yyy is branch No., xxx (e.g.006 ) is RTU type.***
* ***\*"RTU-AC-Load/yyy/xxx", where yyy is branch No., xxx (e.g.007 ) is RTU type.***
* ***\*"RTU-AC-PowerSource/yyy/xxx", where yyy is branch No., xxx (e.g.008 ) is RTU type.***
* ***\*"RTU-PV-AC/DC-PowerSource/yyy/xxx", where yyy is branch No., xxx (e.g.009 ) is RTU type.***

***xxx=000—DC meter without Relay Control; xxx=001—DC meter with Relay Control;***

***xxx=002—AC meter without Relay Control; xxx=003—AC meter with Relay Control;***

***xxx=004—Relay Control only; xxx=005—DC/AC Converter in Serial port;***

***xxx=006—DC Programmable Load in SCPI;***

***xxx=007—AC Programmable Load in SCPI; xxx=008— AC Programmable Source in SCPI;***

***xxx=009— PV Simulator in SCPI; xxx=others—for future development***

***note 2: the definition of*** deviceUID ***is recommended as to: 0x12-xxxx-yyyy-zzzzL, where xxxx is device type, yyyy is locating position and zzzz is index of device.***

### The normal data exchange command/server control command:

The data exchange package from device side in Java format is:

**ByteBuffer**[] dataExchangeDevice = **new** **ByteBuffer**{long deviceUID, int OperatingSerialNo, byte[] data};

The control command package of server in Java format is:

**ByteBuffer**[]controlCommandServer = **new** **ByteBuffer**{long serverUID, int OperatingSerialNo, byte[] command};

***Note: the PDT byte = “0H” in the*** serverUID/deviceUID.

To identify the responses of devices for which command, the server side should keep an Operating-Serial-No variable for each device. The Operating-Serial-No variable increases 1 when a command is sent to the device. The response of the device will accompany with the relevant Operating-Serial-No, so that the server can identify the responses of devices. The Operating-Serial-No is a non-negative integer, that means the Operating-Serial-No variable’s increase-1-process should be (++OperatingSerialNo &= 0x7fffffff;). The negative integer of Operating-Serial-No is reserved for special purpose. For example:

* Operating-Serial-No = -1— the data exchange package of a device is automatically generated by the device in *timeIntervalNormalData* time-interval, using for normal data refresh purpose;
* Operating-Serial-No = -2— the data exchange package of a device is automatically generated by the device in *timeIntervalEnergyData* time-interval, using for energy data refresh purpose;
* Operating-Serial-No = -3— the data exchange package of a device is automatically generated by the device in in *eventReportRelayStatus* relay status being changed, using for relay status event report;
* Operating-Serial-No = -4— the data exchange package of a device is automatically generated by the device in *eventReportSwitchStatus* switch status being changed, using for switch status event report;

### The refresh-period and refresh-command-sets setting command:

To make the control system more flexible, most of devices can work on a data-automatic-refresh mode when their configuration is setting on. In this mode, the device automatically collects data and sends them to BIMG server with a preset time intervals. In default, the time intervals *timeIntervalNormalData = 1000 ms*, *timeIntervalEnergyData = 60000 ms* and the refresh-command-sets are *nulls*. However, the server can change the setting by the following commands.

For short time-interval normal data refresh, with the PDT byte = ‘S’:

**ByteBuffer**[] settingRefreshIntervalComd =

**new** **ByteBuffer**{long deviceUID, long *timeIntervalNormalData*, byte[] *timeRefreshComdBytesN*};

or (for SCPI device)

**ByteBuffer**[] settingRefreshIntervalComd =

**new** **ByteBuffer**{long deviceUID, long *timeIntervalNormalData*, String *timeRefreshComdStrN*};

And for long time-interval energy data refresh, with the PDT byte = ‘L’:

**ByteBuffer**[] settingRefreshIntervalComd =

**new** **ByteBuffer**{long deviceUID, long *timeIntervalEnergyData*, byte[] *timeRefreshComdBytesE*};

or (for SCPI device)

**ByteBuffer**[] settingRefreshIntervalComd =

**new** **ByteBuffer**{long deviceUID, long *timeIntervalEnergyData*, String *timeRefreshComdBytesE*};

Where, if *timeIntervalNormalData* < 0 or *timeIntervalEnergyData* < 0 the refresh-mode will off; if *timeRefreshComdBytesN, timeRefreshComdStrN, timeRefreshComdBytesE, timeRefreshComdBytesE* is omitted the previous refresh-command-sets setting is used.

### The heartbeat message used in communication maintaining process:

In normal condition, the device will automatically send refreshment data or response server control command within a time interval. If the time interval exceeds a threshold (default value is 5000 ms), the device will automatically generate a heartbeat request message with the PDT byte = “H”:

**ByteBuffer**[] requestHeartBeatDevice =

**new** **ByteBuffer**{long deviceUID, long currentDeviceTime, string deviceName};

The server can change the heartbeat threshold setting by the following command with the PDT byte = “H”:

**ByteBuffer**[]settingHeartBeatDevice =

**new** **ByteBuffer**{long serverUID, long newHeartBeatThreshold};

If heartBeatThreshold < 0, the heartbeat is disabled.

***note: to be compatible with Modbus Protocol, the data order of ByteBuffer is defined in ByteOrder.BIGENDIAN.***