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

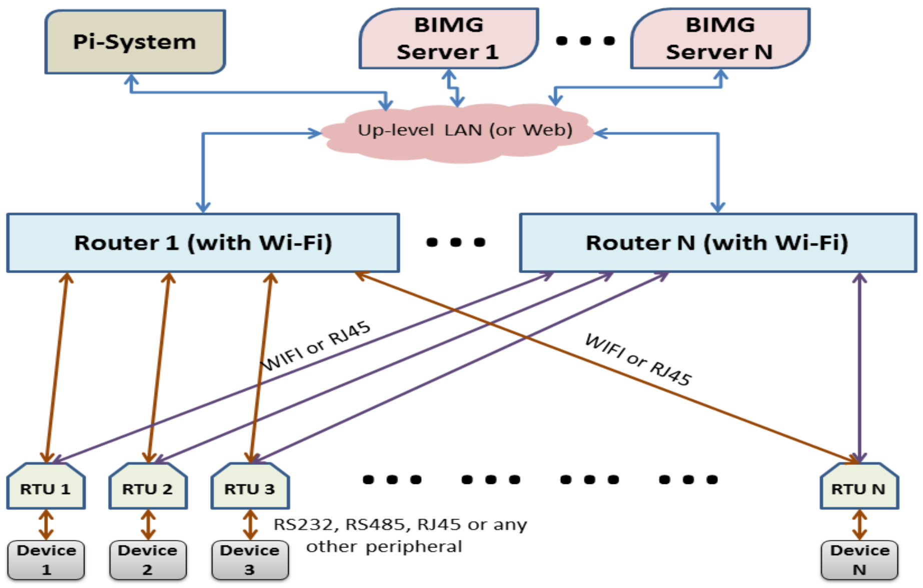
2014.12.30

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 communications channels over a single TCP connection. It is designed to be implemented in web browsers and web servers, 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 is that its handshake is interpreted by HTTP servers as an Upgrade requirement. 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’.

The initial requirement/response commands used in communication connecting process:

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

```
ByteBuffer[] requirementInitialDevice = new ByteBuffer{long deviceUID, string deviceName};
```

The initial response command of server in Java format is (with the PDT byte = ‘I’ or ‘i’):

(with the PDT byte = ‘I’) `ByteBuffer[] responseInitialServer_I =`

```
              new ByteBuffer{ long serverUID, string serverName};
```

`or` (with the PDT byte = ‘i’) `ByteBuffer[] responseInitialServer_i =`

```
              new ByteBuffer{ long serverUID, long serverCurrentTime, string serverName};
```

If the device has used NTP to synchronize its timer, the `responseInitialServer_I` response is to be used. Otherwise the `responseInitialServer_i` response is to be used. The `responseInitialServer_i` initial response of server includes a `serverCurrentTime` parameter, which is taken from the server’s timer when the initial response command is formed. So devices can use this `serverCurrentTime` parameter to correct its timer in the WebSocket initializing process.

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 requirement command. It won’t work until the initial requirement 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 requirement command of device.

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

- ***"RTU-DC-Branch_yy_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.003 ~ 004) is RTU type.***
- ***"RTU-AC_DC-BranchSwitch_yyy_xxx", where yyy is branch No., xxx (e.g. 002) 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=004—AC meter without Relay Control; xxx=003—AC meter with Relay Control;
xxx=002—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 = 'N':

```
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 = ‘E’:

```
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 communication state of device used in communication maintaining:

The device will send its communication state message under the server requirement. The server requirement command message is with the PDT byte = “S”:

```
ByteBuffer[] requirementDeviceCommState =
                new ByteBuffer{long serverUID, string deviceName};
```

The device response message is also with the PDT byte = “S”:

```
ByteBuffer[] responseDeviceCommState =
                new ByteBuffer{long deviceUID, int DeviceCommState, int WSCommState,
                               int PiSCommState, String currentWiFiRouter };
```

where state = 0—start to connect; 2—in normal condition; >0x100—in error or disconnected and PiSCommState is the last error returned from Pi System.

The remote parameter-setting file loading & saving command used in device management:

When a device needs to be re-set its parameter, the maintenance server can send a remote parameter-setting file saving command. The remote parameter-setting file saving command message is with the PDT byte = ‘X’:

```
ByteBuffer[] requirementRemoteSettingFile = new ByteBuffer{long serverUID,
                         byte reBootEnable, byte FileIndex,  String SavingToFileContext};
```

After received the remote parameter-setting file saving command, the device saves the previous file to a back-up file and renew the context with `SavingToFileContext`. The `FileIndex` is used for selecting the setting file, which `FileIndex` can be ‘X’, ‘n’, ‘N’, ‘e’, ‘E’, ‘R’, ‘S’ or ‘1’~‘3’. If reBootEnable = ‘B’ or ‘b’, the device will reboot in order to let the new setting file effective immediately.

If `SavingToFileContext` is null (length = 0), the device doesn’t save a file, but loads the relevant parameter-setting file from its current working directory and sends the file to server. The server may modify (edit) the received data as text, and then send back to the device. In this way, the server side can conveniently re-set the operating parameter of devices.

The system time correction command used for system time synchronization:

When the BIMG system needs to synchronize time, the current master server can send a system time correction command. The system time correction command message is with the PDT byte = ‘T’ or ‘t’:

```
ByteBuffer[] requirementTimeSynchronization =
                      new ByteBuffer{long serverUID, long currentServerTimeMillis};
```

After received the system time correction command, the device synchronizes its time with the parameter `currentServerTimeMillis`.

The server redirecting requirement commands used in system management process:

When the current master server is going to off-line or going to transfer the system control to another on-line working server, the current master server can send the server redirecting requirement command. The server redirecting requirement command can be sent in broadcast mode (when the current master server is going to off-line), or just be sent to an individual device (when the device needs to be transfer to another server for parameter-setting or other kind of management). The server redirecting requirement command message is with the PDT byte = 'R':

```
ByteBuffer[] requirementServerRedirecting =
                             new ByteBuffer{long serverUID, string newMasterServerIP};
```

After received the server redirecting requirement command, the device closes its current WebSocket channel, and then re-connect to the new assigned master server.

*The Pi-System setting command for Pi-System accessing:

The device will re-generate the parameters for Pi-System accessing under the server requirement. The server requirement command message is with the PDT byte = 'P' (for IP4) or 'p' (for IP6):

```
ByteBuffer[] requirementPiSystemSetting =
IP4➔      new ByteBuffer{long serverUID, byte[4] PiSystemURL, string RootElementWebID};
or IP6➔   new ByteBuffer{long serverUID, byte[16] PiSystemURL, string RootElementWebID};
```

**The heartbeat message used in communication maintaining process:

In normal condition, the device automatically sends heartbeat message within a setting time interval to acknowledge server side that the Websocket channel is working properly. And the server should reply the device's heartbeat message with a heartbeat response message of server to finish this kind of handshake data-exchange. The server should send a heartbeat message (whether a heartbeat response or a new one, or any other heartbeat threshold setting commands) back to the device within another setting time interval. Otherwise, the device will the device will automatically close the current websocket channel and re-connect an active server with new websocket channel. The two time thresholds in the device side are defined as heartBeatSendingTimeThreshold and heartBeatReceivingTimeThreshold individually. If the time interval exceeds heartBeatSendingTimeThreshold threshold, the device will generate a heartbeat message with the PDT byte = 'H':

```
ByteBuffer[] requirementHeartBeatDevice =
         new ByteBuffer{long deviceUID, long currentHeartBeatSendingTime, string deviceName};
```

The server replies the device with a heartbeat response message as follow (with PDT byte = 'H' or 'h'):

```
   ByteBuffer[] responseHeartBeatDevice = new ByteBuffer{long serverUID};
```

The server can set the device's two time thresholds by heartbeat threshold setting commands as follows (with PDT byte = 'H' or 'h'):

```
ByteBuffer[] settingHeartBeatDevice1 =
                new ByteBuffer{long serverUID, long newHeartBeatSendingTimeThreshold};
or ByteBuffer[] settingHeartBeatDevice2 = new ByteBuffer{long serverUID,
             long newHeartBeatSendingTimeThreshold, long newHeartBeatReceivingTimeThreshold};
```

The `settingHeartBeatDevice1` only sets the device’s heartBeatSendingTimeThreshold and leave the `newHeartBeatReceivingTimeThreshold` unchanged, while the `settingHeartBeatDevice2` sets both of the two time thresholds: heartBeatSendingTimeThreshold and heartBeatReceivingTimeThreshold. The default values of heartBeatSendingTimeThreshold = 5000 ms and heartBeatReceivingTimeThreshold = 30000 ms. If the time counter exceeds heartBeatReceivingTimeThreshold, the device will trigger the websocket channel’s re-starting. Therefore, the server has the responsibility to send back a heartbeat response message (or any other equivalent handshake data-exchange command) to the device within the interval heartBeatReceivingTimeThreshold.

If the PDT byte = ‘h’, the device is set up to its `heartbeatReplaceMode.` In this mode, any sending message reset HeartBeatSendingTimeCT; any receiving message reset HeartBeatReceivingTimeTC. So it can omit the heartbeat message hand-shaking when the websocket channel is busy. However, the server is responsible for countering the timer whether there is no message from the device within the threshold, and sending a heartbeat response message when there is no message to device within an interval of heartBeatReceivingTimeThreshold.

If heartBeatSendingTimeThreshold < 0, the heartbeat is disabled.

Note 1: the title with * is newly modified for Pi-System re-setting when the Pi-System is changed. And the title with ** is newly modified for BIMG server to detect the failure of the WebSocket channel with devices and help devices to re-construct their the WebSocket channel.

Note 2: to be compatible with Modbus Protocol, the data order of ByteBuffer is defined in ByteOrder.BIGENDIAN.