Z-Stack Linux Gateway Developer's Guide

Version 1.0

Texas Instruments, Inc.
San Diego, California USA

Table of Contents

1. Objective

The Z-Stack Linux Gateway is a complete application development suite that provides an abstracted API set and Linux-based servers that can be used to develop a ZigBee gateway solution.

2. Scope

The intended audience for this document is someone trying to develop their own Z-Stack Gateway application. This document assumes that the reader is already familiar with the Z-Stack Gateway User's Guide, the Gateway Sample Application and some basic ZigBee concepts. The user should also be acquainted with the *Z*-Stack Linux Gateway- Application Programming Interface document.

3. Definitions, Abbreviations, Acronyms

Term	Definition
NPI	Network Processor Interface
ZNP	ZigBee Network Processor
SBL	Serial Bootloader
SDK	Software development kit

4. Building the full source package

The following paragraphs will guide you through the set of instructions required to build the entire source code.

4.1 Setup and build

The script required to rebuild the full package on the Linux PC is included as part of this release.
The sources for the full package are available at <INSTALL>/Source.

In order to rebuild the entire source package, please first install the Linaro tool chain from the following link:

http://software-dl.ti.com/sitara_linux/esd/AM335xSDK/latest/index_FDS.html

Following are the steps to rebuild the full package:
Make sure your PATH and TCLIB variable are pointing to the correct toolchain folder location

```
>export PATH=<LINARO INSTALL LOCATION>/bin/:$PATH
>export TCLIB=<LINARO INSTALL LOCATION>/lib/
```

Navigate to the following directory and launch the build script:
>cd <INSTALL>/Source/
>./build_all

The included pre-compiled binary was built with the GCC 4.7 2013.03 version.

5. Z-Stack Linux Gateway Overview

The Z-Stack Linux Gateway is a collection of software and tools that can be used to create a ZigBee *gateway*. The targeted product architecture is a Linux host and ZigBee Network Processor. The ZigBee Network Processor is a Texas Instruments Z-Stack Network Processor or ZNP. The software, designed to run on the Linux host, is a collection of server applications and scripts that accomplish two key goals:

- Abstraction of the details of ZigBee network management
- Acceleration of application development through a simple and intuitive API

The following block diagram shows a high level overview of the key components of Z-Stack Linux Gateway.

5.1 Gateway SW Architecture

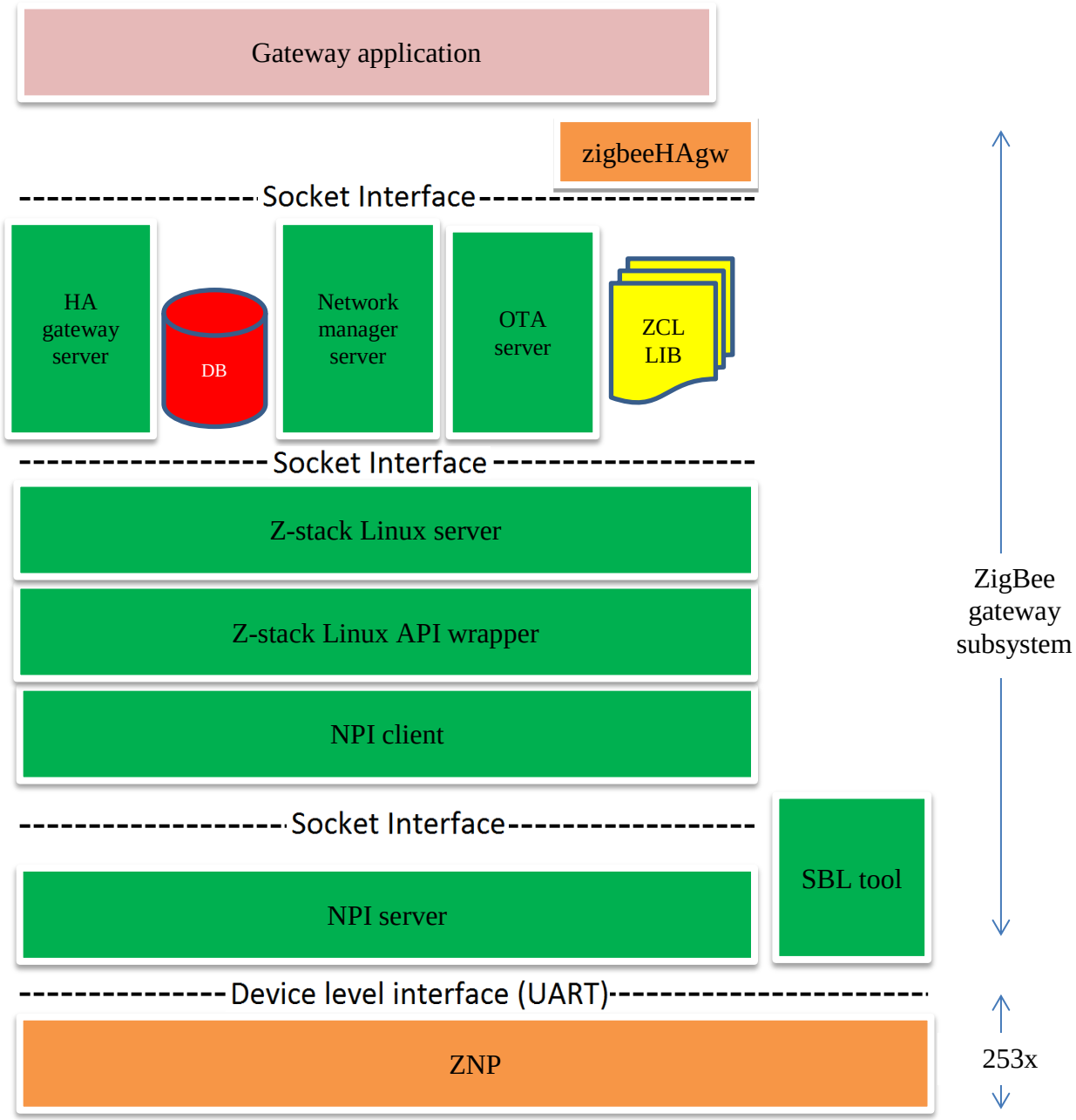

Figure 1: Gateway block diagram

A typical gateway application needs to communicate only with the servers at the top-level i.e., Network Manager, HA Gateway and OTA Upgrade servers. These servers, in turn, communicate with the servers below them via TCP socket. A shell script called *zigbeeHAgw* is used to bring up the various servers and track their status.

The Z-Stack Linux Gateway User Guide provides details about each of these blocks. Users are encouraged to familiarize themselves with each of the blocks before proceeding.

Following is a brief description of the servers that the application connects to directly:

- **Network Manager Server**
  The Network Manager server provides support for management of devices on the network using ZigBee stack services provided by servers below. An application could leverage the APIs exposed by this server, for creating and maintaining a ZigBee network, acquiring and displaying device information, network status etc. For a complete list of APIs implemented by the Network Manager Server, please refer to the Z-Stack Linux Gateway API document Chapter 8.

- **HA Gateway Server**
  The Gateway server provides APIs for communicating with individual or groups of devices, in order to control them, read/change their attributes etc. Chapter 9 of the Z-Stack Linux Gateway API document provides a complete list of APIs supported by the Gateway Server.

- **OTA Upgrade Server**
  The OTA Upgrade Server provides firmware upgrade services for devices in the Gateway's ZigBee network. It provides APIs to manage device upgrade images, and initiate/control the actual upgrade process over-the-air. Chapter 10 of the Z-Stack Linux Gateway API document provides a complete list of APIs supported by the OTA Upgrade Server.

Each of the servers above provides APIs that can be accessed by the application, and a TCP socket connection that the application can connect to. The servers themselves have some interdependencies and need to be brought up in a fixed order. A shell script called ***zigbeeHAgw*** is provided that needs to be run to bring up all the servers in the correct order with the correct parameters. This script then monitors all the servers, and also performs important functions with regard to Power Down, Sleep, Wakeup and Reset. Details on how this script specifically handles Power down, Reset and Sleep functionality are provided in Section 7.

5.2 Gateway Sample Application

The Z-Stack Linux Gateway offering includes a sample Gateway application that is a good starting point for anyone trying to develop their own application. The example, included in the release along with source code, performs basic Gateway functions such as creating the network and adding/controlling common ZigBee devices. It also demonstrates other interesting ZigBee functionality such as creating Groups and Scenes for devices connected to the network.

Section 6 of the Z-Stack Linux Gateway User's Guide provides several details about the sample application, and has instructions on how to exercise its various features. Further sections of this guide will refer to source code in the sample application, the reader is encouraged to install and play around with the sample application and familiarize themselves with the code layout etc. The location of the Gateway application sources is <INSTALL>/Source/Projects/zstack/linux/demo and will henceforth be referred to as <GATEWAY SAMPLE APP>

Please note that this sample application is for demonstration purposes only, and is not intended to be a reference implementation.

6. Protobuf

The Z-Stack Linux Gateway APIs are defined and implemented using a serialization protocol known as Protocol Buffers or Protobuf. Protocol Buffers are a language-independent and platform-neutral way of serializing structured data for efficient data communication.

Typically, APIs are expressed using Protocol Buffers notation and terminology in one or more .proto files. These files can then be compiled into one of several languages supported by Protobuf using a specialized Protobuf compiler. This generates source code that provides data access classes, simple accessors for each field within each message, as well as methods to serialize and parse the whole message structure to/from raw bytes.

More details and download instructions for Protobuf are available here: https://code.google.com/p/protobuf/
Here's a list of current languages supported: https://code.google.com/p/protobuf/wiki/ThirdPartyAddOns

6.1 Protobuf usage in Z-Stack Gateway

APIs for the three servers that need to be accessed by the Gateway application (HA Gateway, Network Manager and OTA servers) are defined in their corresponding .proto files. An application writer can compile these proto files into one of the languages supported by Protobuf by downloading the appropriate language and compiler support. You can find the proto files for the three servers here:

```
<Linux_Host_Install_Folder>/Proto_files:
nwkmgr.proto
gateway.proto
otasrvr.proto
```

The Z-Stack Gateway servers, as well as the sample application are written in C and hence they use a C implementation of the Protobuf format, called Protobuf-c. Running the Protobuf-c compiler on the (above) proto files, generates C source and header files that are built into the servers, and will also be required to be built into the application if it is written in the same language. Details and download instructions for Protobuf-c are available here: https://github.com/protobuf-c/protobuf-c.

The 'C' code generated by the Protobuf-c compiler requires an "engine" to run. This is essentially a library of encoding and decoding routines which also needs to be linked with any executable built with Protobuf generated C code. The Z-Stack Linux Gateway package provides a copy of the libprotobuf-c engine, at this location:

<*INSTALL*>/protobuf/libprotobuf-c.so.0

The Protobuf-c generated files for the 3 servers are also included with the release:

<*INSTALL*>/Source/Projects/zstack/linux
nwkmgr/nwkmgr.pb-c.*
hagateway/gateway.pb-c.*
otaserver/otasrvr.pb-c.*

Note, the libprotobuf-c.so.0 library and the Protobuf-c generated files are useful only if you intend to implement your application in "C" like the Gateway sample application.

6.1.1 Protobuf Encoding/Decoding Example

Whenever an application sends a command to one of the servers, it needs to first call a fixed set of functions to "Protobuf-encode" the message. Similarly, when it receives a command from the server, it needs to call a fixed set of functions to decode it.

For instance, say your application wants to send a command to the Gateway to determine the status of the network. There's a Network Manager API for it called **NWK_ZIGBEE_NWK_INFO_REQ** (Z-Stack Linux Gateway API document, Section 8.2.2). Your application would then construct a message to send to the Network Manager server as shown here:

<GATEWAY SAMPLE APP>/engines/network_info_engine.c:

```
129 void nwk_send_info_request(void)
130 {
131         pkt_buf_t * pkt = NULL;
132         uint8_t len = 0;
133         NwkZigbeeNwkInfoReq msg = NWK_ZIGBEE_NWK_INFO_REQ__INIT;
134
135         UI_PRINT_LOG("nwk_send_info_request: Sending NWK_INFO_REQ__INIT");
136
137         len = nwk_zigbee_nwk_info_req__get_packed_size(&msg);
138         pkt = malloc(sizeof(pkt_buf_hdr_t) + len);
139
140         if (pkt)
141         {
142                 pkt->header.len = len;
```



```
143                 pkt->header.subsystem =
Z_STACK_NWK_MGR_SYS_ID_T__RPC_SYS_PB_NWK_MGR;
144                 pkt->header.cmd_id =
NWK_MGR_CMD_ID_T__NWK_ZIGBEE_NWK_INFO_REQ;
145
146                 nwk_zigbee_nwk_info_req__pack(&msg, pkt-
>packed_protobuf_packet);
```

The lines highlighted above show the three APIs that are called to initialize a message object (`<>__INIT`), determine its length (`<>__get_packed_size`), and then pack or encode it (`<>_req__pack`) into the actual message that is then sent across the socket interface. Of course, there are several lines of code in between, where the memory for the message is allocated, and the message is populated according to the parameters defined in the API document. You also need to free any memory allocated for the message, after it has been sent over the socket connection.

Similarly, every time a command is received from a particular server, APIs will need to be called to unpack and decode the message. In the same file look at the following lines that demonstrate how a response to the message above is handled:

```
 55 void nwk_process_ready_ind(pkt_buf_t * pkt)
 56 {
 57         NwkZigbeeNwkReadyInd *msg = NULL;
 58
 59         if (pkt->header.cmd_id !=
NWK_MGR_CMD_ID_T__NWK_ZIGBEE_NWK_READY_IND)
 60         {
 61                 return;
 62         }
 63
 64         UI_PRINT_LOG("nwk_process_ready_ind: Received
NWK_ZIGBEE_NWK_READY_IND");
 65
 66         msg = nwk_zigbee_nwk_ready_ind__unpack(NULL, pkt->header.len, pkt-
>packed_protobuf_packet);
 67
 68         if (msg)
 69         {
 70                 ds_network_status.state = ZIGBEE_NETWORK_STATE_READY;
 71                 ds_network_status.nwk_channel = msg->nwkchannel;
 72                 ds_network_status.pan_id = msg->panid;
 73                 ds_network_status.ext_pan_id = msg->extpanid;
 74                 ds_network_status.permit_remaining_time = 0x0;
 75                 ds_network_status.num_pending_attribs = 0x0;
 76
 77                 nwk_zigbee_nwk_ready_ind__free_unpacked(msg, NULL);
```

This type of pack/unpack functions are called every time a command is sent or received by the Gateway application. All the highlighted functions are declared and defined in the Protobuf-c compiler generated files for the Network Manager, namely nwkmgr.pb-c.h, nwkmgr.pb-c.c. Generated C-files are included for all the servers that the application will communicate with.

6.1.2 Packet Header information

The actual data packet sent over the application interface includes more than just the Protobuf-encoded message (seen above). A 4-byte header is expected to precede the Protobuf-packed message, and it needs to include the following information:

- **len** – 16bit number that specifies the length (in bytes) of the protobuf-packed packet.
- **Subsystem** – 1 byte ID of subsystem to/from which the packet is sent/received. It can be either:
  - **18** - Network Manager Server
  - **19** - Gateway Server
  - **20** - OTA Server

- **cmd_id** – 1 byte command ID of the actual command being sent. This value is also available inside the packed message. The actual command ID numbers are provided in the protobuf definition files (.proto files) that are part of the Z-Stack Linux Gateway package. When using command IDs in your code, always use the defined names (never hardcode the command ID numbers), as the numbers may change between releases.

Reviewing the same file we did before, the highlighted lines below show memory being allocated for the encoded message *and* this 4-byte header, and then the fields of the header being populated before the packet is sent to the socket interface function:

<GATEWAY SAMPLE APP>/engines/network_info_engine.c:

```
137         len = nwk_zigbee_nwk_info_req__get_packed_size(&msg);
138         pkt = malloc(sizeof(pkt_buf_hdr_t) + len);
139
140         if (pkt)
141         {
142                 pkt->header.len = len;
143                 pkt->header.subsystem =
Z_STACK_NWK_MGR_SYS_ID_T__RPC_SYS_PB_NWK_MGR;
144                 pkt->header.cmd_id =
NWK_MGR_CMD_ID_T__NWK_ZIGBEE_NWK_INFO_REQ;
145
146                 nwk_zigbee_nwk_info_req__pack(&msg, pkt-
>packed_protobuf_packet);
147
148                 if (si_send_packet(pkt,
(confirmation_processing_cb_t)&nwk_process_info_cnf, NULL) != 0)
149                 {
```

As mentioned in the packet-header description above, all the constants used to populate the packet header, come from the Protobuf-generated header file for the Network manager server:

<INSTALL>/Source/Projects/zstack/linux
nwkmgr/nwkmgr.pb-c.h:

```
 49 typedef enum _ZStackNwkMgrSysIdT {
 50   Z_STACK_NWK_MGR_SYS_ID_T__RPC_SYS_PB_NWK_MGR = 18
 51 } ZStackNwkMgrSysIdT;

 52 typedef enum _NwkMgrCmdIdT {
 53   NWK_MGR_CMD_ID_T__ZIGBEE_GENERIC_CNF = 0,
 54   NWK_MGR_CMD_ID_T__ZIGBEE_GENERIC_RSP_IND = 1,
 55   NWK_MGR_CMD_ID_T__NWK_ZIGBEE_SYSTEM_RESET_REQ = 2,
 56   NWK_MGR_CMD_ID_T__NWK_ZIGBEE_SYSTEM_RESET_CNF = 3,
 57   NWK_MGR_CMD_ID_T__NWK_ZIGBEE_SYSTEM_SELF_SHUTDOWN_REQ = 4,
 58   NWK_MGR_CMD_ID_T__NWK_SET_ZIGBEE_POWER_MODE_REQ = 5,
 59   NWK_MGR_CMD_ID_T__NWK_SET_ZIGBEE_POWER_MODE_CNF = 6,
 60   NWK_MGR_CMD_ID_T__NWK_GET_LOCAL_DEVICE_INFO_REQ = 7,
 61   NWK_MGR_CMD_ID_T__NWK_GET_LOCAL_DEVICE_INFO_CNF = 8,
 62   NWK_MGR_CMD_ID_T__NWK_ZIGBEE_NWK_READY_IND = 9,
 63   NWK_MGR_CMD_ID_T__NWK_ZIGBEE_NWK_INFO_REQ = 10,
```

Similar values for subsystem IDs and command IDs can be found in the respective Protobuf- generated headers of the other servers as well.

7. Overview of Gateway Application

This section attempts to breakdown a typical Gateway Application into its essential functional blocks and goes into detail in each one.

Code snippets referred to in subsequent sections will be from the sample Gateway application and in the "C" programming language. However the concepts covered should be applicable to any language in which you choose to implement your application.

7.1 Connection to the servers

Before an application can send and receive commands to/from the servers, it needs to connect to them. The ZigBee gateway software supports one active client application at a time, and that application should be connected to all the three servers for proper operation. Let's take a look at how a typical Gateway application might do that.

All the three servers (Network Manager, HA Gateway and OTA Upgrade) have separate TCP socket connections and service access point interfaces. Typically, an application would need the IP address and port number for each of the servers it needs to connect to. The servers all have a fixed port number they use, and this number is displayed for each server when the servers are launched (via the *zigbeeHAgw* script). The function in charge of making and then maintaining a connection to the servers, should use this information (IP address and port number) to create a connection to each of the servers. You can refer to the following functions for an example of how to populate the server details and create a socket connection:

<GATEWAY SAMPLE APP>/framework/tcp_client.c:

```
 78    int tcp_new_server_connection(server_details_t * server_details, char *
              hostname, u_short port, server_incoming_data_handler_t
              server_incoming_data_handler, char * name,
              server_connected_disconnected_handler_t
              server_connected_disconnected_handler)

 110   int tcp_connect_to_server(server_details_t * server_details)
```

In the above functions, typical calls to Socket APIs such as *connect()* and *socket()* are made to establish connection with each of the servers. The file descriptors returned are used to populate the *server_details* data structures shown below. This structure also contains pointers to callback functions to handle socket connection/disconnection and incoming data:

<GATEWAY SAMPLE APP>/framework/socket_interface.c:

```
 97 server_details_t network_manager_server;
 98 server_details_t gateway_server;
 99 server_details_t ota_server;
```

With the structure defined as follows:

<GATEWAY SAMPLE APP>/framework/tcp_client.h

```
 53 typedef void (* server_incoming_data_handler_t)(void * buf, int len);
 54 typedef void (* server_connected_disconnected_handler_t)(void);
 55
 56 typedef struct
 57 {
 58         struct sockaddr_in serveraddr;
 59         server_incoming_data_handler_t server_incoming_data_handler;
 60         int fd_index;
 61         tu_timer_t server_reconnection_timer;
 62         char * name;
 63         server_connected_disconnected_handler_t
server_connected_disconnected_handler;
 64         bool connected;
 65         int confirmation_timeout_interval;
 66 } server_details_t;
```

The variable *fd_index* points to a single entry into an array of descriptors. Each entry holds information about a particular server connection such as the file descriptor that represents the socket connection, callback

functions/handlers that handles incoming data on the socket. A lot of these details are specific to the sample application's implementation, and can/should be modified suitably for the user's own application.

An important requirement for the Gateway application is to constantly monitor server connection status. The application *should* expect to lose connection with the servers in certain scenarios, such as when a System Reset command is issued by the application (via the NWK_ZIGBEE_SYSTEM_RESET_REQ command; for details, see Z-Stack Linux Gateway API document – Section 8.1.1). An application should constantly attempt to reconnect to the servers whenever it detects a disconnection.

In the same file referenced above, you can look at how a disconnect event is handled in the sample application in the following functions:

<GATEWAY SAMPLE APP>/framework/tcp_client.c:

```
159    void tcp_socket_event_handler(server_details_t * server_details)
...
...
173         else if (remaining_len == 0)
174         {
175                 UI_PRINT_LOG("Server %s disconnected", server_details-
>name);
176                 close(polling_fds[server_details->fd_index].fd);
177                 polling_undefine_poll_fd(server_details->fd_index);
178                 server_details->connected = false;
179
180                 if (server_details->server_connected_disconnected_handler
!= NULL)
181                 {
182                         server_details-
>server_connected_disconnected_handler();
183                 }
184
185                 ui_redraw_server_state();
186                 tcp_socket_reconnect_to_server(server_details);
```

In the event of a disconnection with a particular server, the above function closes the socket connection, calls a 'connection/disconnection handler', which is a callback function registered specifically for that server, and then immediately attempts to connect back to the server. The connection handler could do things like clearing out internal saved state, refresh display etc. You can look at the connection handler for all the servers here:

<GATEWAY SAMPLE APP>/framework/socket_interface.c

```
523 void nwk_mgr_server_connected_disconnected_handler(void)

542 void gateway_server_connected_disconnected_handler(void)

548 void ota_server_connected_disconnected_handler(void)
```

In most cases *zigbeeHAgw* script brings the servers back up in case one or more of them exit. But in certain situations, for instance when the application issues a shutdown (via the NWK_ZIGBEE_SELF_SHUTDOWN command), the servers will not come back up, and it is up to the application to handle that scenario.

7.2 Sending commands to the Servers

Once connection is established with the servers, the next step is to send commands to the servers and process received commands. Details of all the commands that can be sent to/received from the servers are available in the Z-Stack Linux Gateway API document. Details of the C-structures that are used to represent these individual commands are in their corresponding Protobuf generated header files.

In our sample application, after establishing the connection to the servers, the state machine of the application is initiated and the first command is sent to the network manager to obtain network status information: NWK_MGR_CMD_ID_T__NWK_ZIGBEE_NWK_INFO_REQ. This is handled in the following function called up the application:

<GATEWAY SAMPLE APP>/engines/network_info_engine.c:

```
129 void nwk_send_info_request(void)
```

We have already covered details of how the protobuf message is initialized and packed in Section 5.1.1. The structure of the entire packet that is sent over the socket connection is here:

<GATEWAY SAMPLE APP>/framework/types.h:

```
 96 typedef struct
 97 {
 98         uint16_t len;
 99         uint8_t subsystem;
100         uint8_t cmd_id;
101 } pkt_buf_hdr_t;
102
103 typedef struct
104 {
105         pkt_buf_hdr_t header;
106         uint8_t packed_protobuf_packet[];
107 } pkt_buf_t;
```

In *pkt_buf_t,* the first field is the packet header that is populated with information regarding the length of the protobuf message, the subsystem (or server) for which this message is intended, and it's unique command ID. Then the protobuf message is encoded or packed into the *packed_protobuf_packet* array that follows the header:

<GATEWAY SAMPLE APP>/engines/network_info_engine.c:

```
 129 void nwk_send_info_request(void)
 130 {
 131         pkt_buf_t * pkt = NULL;
 132         uint8_t len = 0;
 133         NwkZigbeeNwkInfoReq msg = NWK_ZIGBEE_NWK_INFO_REQ__INIT;
 134
 135         UI_PRINT_LOG("nwk_send_info_request: Sending NWK_INFO_REQ__INIT");
 136
 137         len = nwk_zigbee_nwk_info_req__get_packed_size(&msg);
 138         pkt = malloc(sizeof(pkt_buf_hdr_t) + len);
 140         if (pkt)
 141         {
 142                 pkt->header.len = len;
 143                 pkt->header.subsystem =
Z_STACK_NWK_MGR_SYS_ID_T__RPC_SYS_PB_NWK_MGR;
 144                 pkt->header.cmd_id =
NWK_MGR_CMD_ID_T__NWK_ZIGBEE_NWK_INFO_REQ;
 145
 146                 nwk_zigbee_nwk_info_req__pack(&msg,
 pkt->packed_protobuf_packet);
 147
 148                 if (si_send_packet(pkt,
(confirmation_processing_cb_t)&nwk_process_info_cnf, NULL) != 0)
 149                 {
 150                         UI_PRINT_LOG("nwk_send_info_request: Error: Could
not send msg");
 151                 }
 152
 153                 free(pkt);
```


The last highlighted line shows the call made to send the packet over the socket interface. The *si_send_packet* function has the following signature:

<GATEWAY SAMPLE APP>/framework/socket_interface.c

```
 229 int si_send_packet(pkt_buf_t * pkt, confirmation_processing_cb_t
_confirmation_processing_cb, void * _confirmation_processing_arg)
```

The first argument is a pointer to the packet itself, the second is a call back function for processing the *confirmation* for the message being sent, and the third is the optional argument for this function. Whenever a request-type command is sent to a server, a confirmation message is sent back to the application by the server in a synchronous manner. The gateway application is not supposed to send out a new outgoing request until it receives the confirmation for the previous request.

In the *si_send_packet()* function, the header is first checked to determine which server the message is meant for, the server details for that server are then extracted, and then (if connection is still intact) the following function is called to send the packet over the respective connection:

<GATEWAY SAMPLE APP>/framework/tcp_client.c

```
  68 int tcp_send_packet(server_details_t * server_details, uint8_t * buf, int
len)
  69 {
  70         if (write(polling_fds[server_details->fd_index].fd, buf, len) !=
len)
  71         {
  72                 return -1;
  73         }
  74
  75         return 0;
  76 }
```

The server that receives this message expects to receive a 4 byte header followed by the actual protobuf encoded message. The length parameter supplied when making the *write()* call includes the length of the protobuf message as well as the (4 byte) header.

After sending the packet over the socket interface, the *si_send_packet()* function sets up some internal state variables to indicate that the application is currently waiting for a confirmation for the command it just sent out. Note: the application will not (and should not) process any other commands till it receives the confirmation for the one just sent out. A timer is used in this application to timeout if the confirmation isn't received within a stipulated time period.

7.3 Handling incoming Commands from the Servers

There are several different kinds of messages that an application may receive from the servers. It could receive a confirmation message for a command it recently sent out, it could receive a response for a command sent out earlier that solicited some information from a ZigBee device on the network, or it could receive an unsolicited incoming indication due to some network activity. (Details on the different kind of messages supported by the servers are available in the Z-Stack Linux Gateway API document, Chapter 4). In the sample application, all these different types of incoming messages are handled in the same way. Your application may choose to handle them differently.

In steady state, the sample application polls for activity on the set of file descriptors that represent the various connections to the servers using the *poll()* call. Whenever it receives data (corresponding to a received message) on one of the descriptors, it calls the socket event handler for that particular descriptor. In the sample application, this function handles incoming messages on the socket interface:

<GATEWAY SAMPLE APP>/framework/tcp_client.c

```
 159 void tcp_socket_event_handler(server_details_t * server_details)
```


This function calls *recv()* on the file descriptor corresponding to the socket connection, and then calls the corresponding handler function and passes it the entire packet. The handler functions for all the servers are available here:

<GATEWAY SAMPLE APP>/framework/socket_interface.c

```
377 void si_nwk_manager_incoming_data_handler(pkt_buf_t * pkt, int len)

405 void si_gateway_incoming_data_handler(pkt_buf_t * pkt, int len)

443 void si_ota_incoming_data_handler(pkt_buf_t * pkt, int len)
```

These functions all look at the command ID in the header of the received packet, and then decide how to handle it. Going back to the example in the last section, where the application sends a NWK_MGR_CMD_ID_T__NWK_ZIGBEE_NWK_INFO_REQ to the Network Manager server, it immediately receives a confirmation response from it. The *si_nwk_manager_incoming_data_handler()* (listed above) knows to call the function that was registered to handle this confirmation, i.e., *nwk_process_info_cnf*().In this function, the message is unpacked/decoded as described in Section 5.1.1. The status of the received response is extracted from the message, and can be used to update the internal state of the application and/or refresh the UI display.

<GATEWAY SAMPLE APP>/engines/network_info_engine.c

```
 87 void nwk_process_info_cnf(pkt_buf_t * pkt, void * cbarg)
 88 {
 89         NwkZigbeeNwkInfoCnf *msg = NULL;
 90
 91         if (pkt->header.cmd_id !=
NWK_MGR_CMD_ID_T__NWK_ZIGBEE_NWK_INFO_CNF)
 92         {
 93                 return;
 94         }
 95
 96         UI_PRINT_LOG("nwk_process_info_cnf: Received
NWK_ZIGBEE_NWK_INFO_CNF");
 97
 98         msg = nwk_zigbee_nwk_info_cnf__unpack(NULL, pkt->header.len, pkt-
>packed_protobuf_packet    );
 99
100         if (msg)
101         {
102                 UI_PRINT_LOG("msg->status = %d", msg->status);
103
104                 /* Update network info structure with received information
*/
105                 if (msg->status == NWK_NETWORK_STATUS_T__NWK_UP)
106                 {
107                         ds_network_status.state =
ZIGBEE_NETWORK_STATE_READY;
108                         ds_network_status.nwk_channel = msg->nwkchannel;
109                         ds_network_status.pan_id = msg->panid;
110                         ds_network_status.ext_pan_id = msg->extpanid;
111
```

7.4 Handling user input

Once a Gateway application comes up and has established network status etc, it can then start to accept input from user to start interacting with the network and/or devices.

The sample application is a terminal application with a menu-based UI that accepts user input via short-cut keys, or by navigating to the desired item (via arrow keys) and then selecting it using the return key. Most of the UI handling is done in the following functions:

<GATEWAY SAMPLE APP>/framework/user_interface.c

```
2073 int ui_init(char * log_filename)

435 void console_event_handler(void * arg)
```

The function *ui_init()* initializes the UI for the application, and the *console_event_handler()* handles user-input.

As an example, one of the first commands that a user is expected to send after starting the application, is to open the network up for ZigBee devices to join (via the NWK_SET_PERMIT_JOIN_REQ command, see Z-Stack Linux Gateway API document – Section 8.2.4). As an argument to this function, the user also specifies the amount of time for which to keep the network open. You can see the *console_event_handler()* function handle this user event:

<GATEWAY SAMPLE APP>/framework/user_interface.c

```
1074        switch (current_action)
1075        {
1076                case ACTION_PRMT_JOIN:
1077                        comm_send_permit_join(action_value[current_action]);
1078                        break;
```

In the *comm_send_permit_join()* function the corresponding Protobuf message is constructed and sent over the socket interface:

<GATEWAY SAMPLE APP>/engines/commissioning_engine.c:

```
 239                 requested_join_time = joinTime;
 240
 241                 nwk_set_permit_join_req__pack(&msg, pkt-
>packed_protobuf_packet);
 242
 243                 if (si_send_packet(pkt, &comm_process_permit_join, NULL)
!=0 )
```

In the confirmation callback function *comm_process_permit_join() (*in the same file), an internal data structure *ds_network_status* is updated and a timer is set up that is triggered every second.

```
 78
 90         msg = nwk_zigbee_generic_cnf__unpack(NULL, pkt->header.len,
pkt->packed_protobuf_packet);
 91
 92         if (msg)
 93         {
 94                 if (msg->status == NWK_STATUS_T__STATUS_SUCCESS)
 95                 {
 96                         UI_PRINT_LOG("comm_process_permit_join: Status
SUCCESS.");
 97
 98                         ds_network_status.permit_remaining_time =
requested_join_time;
 99
100                         UI_PRINT_LOG("comm_process_permit_join: Requested
join time %d",
101                         requested_join_time);
102
103                         if ((requested_join_time > 0) &&
(requested_join_time < 255))
104                         {
```



```
105                                     tu_set_timer(&pj_timer, 1000, true,
&comm_permit_join_timer_handler, NULL);
```

In the timer callback function (*comm._permit_join_timer_handler()*), display data is updated, and the *ui_refresh_data()* function is called. On every refresh call, *ds_network_status* is read along with other data structures to update the UI display:

<GATEWAY SAMPLE APP>/engines/commissioning_engine.c:

```
 63 void comm_permit_join_timer_handler(void * arg)
 64 {
 65         if ((ds_network_status.permit_remaining_time == 0) ||
(ds_network_status.permit_remaining_time == 255))
 66         {
 67                 tu_kill_timer(&pj_timer);
 68         }
 69
 70         ui_refresh_display();
 71
 72         if ((ds_network_status.permit_remaining_time > 0) &&
(ds_network_status.permit_remaining_time < 255))
 73         {
 74                 ds_network_status.permit_remaining_time--;
 75         }
 76 }
```

On careful inspection of the sample application code, you will note that the application does *not* translate user input into an actual command to the server unless it has received a confirmation for previously sent command. Only if a confirmation has been received, or a timeout has occurred, will new user input be processed (See *si_send_packet()* in <GATEWAY DEMO APP>/framework/socket_interface.c).

The way user input is handled is very specific to the actual application implementation. A real-world application is expected to have a lot more complexity than the sample application included with this release.

7.5 Typical API Flow/User, Application and Device interaction

Let's assume the servers are up and running (using the *zigbeeHAgw* script), and the application has successfully connected to the servers (Section 6.1). This section walks through the typical flow of APIs from Gateway application to Gateway servers and vice versa during different stages of the application's lifecycle. The application should be equipped to send and receive these commands and handle them appropriately as described in the sections above.

Legend:
-> (Outgoing) Command sent by the Gateway application **to** one of the Servers.
<- (Incoming) Message/Indication received by the application, sent to it by one of the Servers.

Note: The list below omits the Generic Confirmations that are sent in response to most commands. Details of the APIs and their arguments are available in Z-Stack Linux Gateway API document. This section refers to them by name, and also includes a brief description.

At startup:	List of APIs that might be called by a Gateway application at start-up, before accepting any user input
-> NWK_ZIGBEE_NWK_INFO_REQ	Called by the application to request status of the network.
<- NWK_ZIGBEE_NWK_INFO_CNF	Response to the above API, informing the application whether the network is up and running yet or not.
<- NWK_ZIGBEE_NWK_READY_IND	Informs the application of the status of the newly formed network. Includes information about the network such as the PAN ID, and the channel ID. This information can be saved and used to update the UI.
-> NWK_GET_DEVICE_LIST_REQ	Called by application to get a list of devices already in the

	network. Is called once network readiness is established by previous messages.
<- NWK_GET_DEVICE_LIST_CNF	Confirmation for above command, contains list of devices Currently part of network, and their details. Will be empty the first time network is created.
-> NWK_DEVICE_LIST_MAINTENANCE_REQ	This is a request to discover services for one more devices in the network.
<- NWK_ZIGBEE_DEVICE_IND	Informs the application of a new device having joined, or removed from the network, or if device details have changed (in response to message above). Lets the application present the list of actions that can be performed on available devices in the network.

On user input:	List of APIs that the Gateway application may call in response to user-requests.
-> NWK_SET_PERMIT_JOIN_REQ	On user-input, application can instruct Network manager to open the network up for new devices to join.
<- NWK_ZIGBEE_DEVICE_IND	Informs the application that a new device has joined. Information returned can be used to update the UI with the new device that joined the network.
-> DEV_SET_LEVEL_REQ	On user-input, application can call Gateway manager to set a level-control on one more more devices in network.
-> DEV_GET_LEVEL_REQ	Application can call this Gateway manager API to request current level of one/more devices. Can be used to update display.
<- DEV_GET_LEVEL_RSP_IND	Informs the application of current level of a device, application can use this information to update display.
-> DEV_SET_ONOFF_STATE_REQ	On user-prompt application can send this command to switch a device on or off.
-> DEV_GET_POWER_REQ	Command used to request the power readings of a device.
<- DEV_GET_POWER_RSP_IND	Response from the server to the application with the power value and/or status.
-> NWK_SET_BINDING_ENTRY_REQ	On user-input, application can call Network manager to request a binding to be made between two remote devices such as a light and a switch.
<- NWK_SET_BINDING_ENTRY_RSP_IND	Response received by the application that conveys the status of the binding request.
-> GW_ADD_GROUP_REQ	On user-input, application may send request to Gateway server to associate one/more with a group id.
->GW_STORE_SCENE_REQ	Application can call this Gateway API to associate a particular scene for a group of devices, with an ID.
-> GW_RECALL_SCENE_REQ	On user-input, application can issue this request to recall the scene associated with a particular ID.
-> GW_SET_ATTRIBUTE_REPORTING_REQ	Application can use this API to configure reporting of attributes for a particular device in the network. The list of attributes, and interval at which to send the report can be specified as argument.
<- GW_SET_ATTRIBUTE_REPORTING_RSP_IND	

	Response received for the call above, with status. Information returned can be used to update display.
-> OTA_UPDATE_IMAGE_REGISTERATION_REQ	Whenever a new upgrade image is available for one/more devices, application can send this command to register it with the OTA server.
-> OTA_UPDATE_ENABLE_REQ	Application can enable OTA Upgrades using this command
<- OTA_UPDATE_ENABLE_CNF	Response received by the application.
-> NWK_ZIGBEE_SYSTEM_RESET_REQ	On user-input the application can send a hard or a soft reset request to the Network Manager that causes all the servers to reset.
<- NWK_ZIGBEE_SYSTEM_RESET_CNF	Confirmation sent to the application when the servers come back up again. Please note that the application will lose connection with the servers after the RESET_REQ, and before the response is received.

Steady state or Device activity:	Incoming APIs to the application due to device activity and corresponding responses.
<- NWK_ZIGBEE_DEVICE_IND	Informs application that a device has joined/been removed/has changed its properties etc. Can be used to update the display of devices and their information.
<- GW_ATTRIBUTE_REPORTING_IND	Informs application of attribute values for devices for which reporting was configured. Would typically be used to update display and/or take user-defined action.
<- GW_ALARM_IND	Informs application that an alarm has been generated for a particular cluster. The application would need to identify the reason for alarm and notify the user accordingly.
<- DEV_ACE_ARM_REQ_IND	Informs application that an ACE device has been armed.
-> DEV_ACE_ARM_RSP	
-> NWK_SET_ZIGBEE_POWER_MODE_REQ	Application can send this command to the Network Manager to put the Network in "SLEEP" mode. This same API is used to "WAKE" it up when required.
<- NWK_SET_ZIGBEE_POWER_MODE_CNF	Response from Network manager to the above call.
<- OTA_UPDATE_DOWNLOAD_FINISHED_IND	Informs application that a particular device has finished upgrading to the new image hosted by it.

<Add more ?? <- DEV_ZONE_ENROLMENT_REQ_IND -> DEV_ZONE_ENROLMENT_RSP -> DEV_ZONE_STATUS_CHANGE_IND>

Wind down:	APIs called during shutdown of Gateway application.
-> NWK_ZIGBEE_SYSTEM_SELF_SHUTDOWN_REQ	Called by the application to shut down the Gateway subsystem. Servers are disconnected in response.

Please note, that this is just a representative set of APIs that you might expect an application to call/receive from the Gateway servers. The complete list of APIs is documented in the Z-Stack Linux Gateway API document.

7.6 Sequence Numbers

For each command mentioned in the Z-stack Linux Gateway API document, information is also included about the kind of messages the servers generate in response to them. For e.g.,

> **NWK_ZIGBEE_NWK_INFO_REQ**
> *Command type:* **No-response command** / **NWK_ZIGBEE_NWK_INFO_CNF**

The commands are first classified as *Outgoing* (Application -> Subsystem) or *Incoming* (Gateway subsystem -> Application), and then sub-classified by the type of responses they elicit from the servers (Details in API document, Chapter 4).

As mentioned before, a Gateway application can receive both synchronous messages ("Confirmation message"), in response to commands it sends out, and asynchronous messages ("Responses" or "Indications"), due to network activity or due to servers collating information from multiple over-the-air calls before replying back to the application. Each outgoing command is immediately followed by a confirmation message in a synchronous manner. Afterward it may or may not receive an asynchronous Response from the server, depending on the type of command. In order to allow the application to easily correlate an incoming Response with a message it sends out, the APIs support a parameter called **sequence number**.

If a command sent out by an application is expected to generate an incoming response, then the confirmation message it receives from the server (immediately after sending out the command) includes a 'sequence number'. An incoming indication that arrives later in response to this command also includes this same sequence number. This sequence number is a rolling 16-bit unsigned integer that can be used to correlate incoming indications with the commands that are sent out. An application could 'remember' the sequence number received as part of the confirmation message, and use it to process the incoming response/indication that might come several seconds or commands later.

The sample application provides an example of how these sequence number might be tracked. Consider a scenario where a user might want to switch a ZigBee light device ON or OFF, and present the current state of the device on the UI. The application can respond to user-action by sending a DEV_SET_ONOFF_STATE_REQ command (Z-Stack Linux Gateway API document, Section 9.7.2) to the Gateway server. This command will be responded to by a "Confirmation message" from the Gateway server that includes a sequence number. Eventually a Generic Response Indication is sent by the server which will include status information regarding this command and also the same sequence number. Information in this indication can be used to update the UI. The code snippet below shows the function *act_set_on_off_cnf()*, the callback function called when confirmation for the DEV_SET_ONOFF_STATE_REQ command is received:

<GATEWAY SAMPLE APP>/engines/action_engines.c:

```
117         msg = gw_zigbee_generic_cnf__unpack(NULL, pkt->header.len, pkt-
>packed_protobuf_packet);
118
119         if (msg)
120         {
121                 if (msg->status == GW_STATUS_T__STATUS_SUCCESS)
122                 {
123                         UI_PRINT_LOG("act_process_set_onoff_cnf: Status
SUCCESS.");
124
125                         if (addr->ieee_addr != 0)
126                         {
127                                 UI_PRINT_LOG("act_process_set_onoff_cnf:
seq_num=%d", msg->sequencenumber);
128
129                                 cluster_id = ZCL_CLUSTER_ID_GEN_ON_OFF;
130                                 attribute_list[0] = ATTRID_ON_OFF;
131                                 sr_register_attribute_read_request(msg-
>sequencenumber, addr, cluster_id, attribute_list, 1);
```

When a successful return status is received in the confirmation message, the function *sr_register_attribute_read_request()* is called and the sequence number received in this confirmation is passed to it along with other information about the command. As seen below, this function stores the

sequence number in a table, where it stays until a corresponding Response indication is received, or until this table entry times out:

<GATEWAY SAMPLE APP>/engines/state_reflector.c:

```
131                         pending_attribs[free_index].valid = true;
132                         pending_attribs[free_index].sequence_num = seq_num;
133                         pending_attribs[free_index].ieee_addr = addr-
>ieee_addr;
134                         pending_attribs[free_index].endpoint_id = addr-
>endpoint;
135                         pending_attribs[free_index].cluster_id =
cluster_id;
136                         pending_attribs[free_index].num_attributes =
attr_num;
137                         pending_attribs[free_index].timer_val =
READ_ATTR_TIMEOUT_VAL;
138
139                         for (j = 0; j < attr_num; j++)
140                         {
141                                 pending_attribs[free_index].attr_id[j] =
attr_ids[j];
142
UI_PRINT_LOG("sr_register_attribute_read_request: Adding attribute read for
attr_id 0x%    x", pending_attribs[i].attr_id[j]);
143                         }
144
145                         ds_network_status.num_pending_attribs++;
146
147                         if (ds_network_status.num_pending_attribs == 1)
148                         {
149                                 tu_set_timer(&aging_timer, TIMER_CHECK_VAL
* 1000, true, &aging_engine,
150                                 NULL);
151                         }
```

The highlighted lines above show the sequence number along with other information being stored in a table, and a timer being configured that allows these messages to eventually age and timeout. The timer ensures that the table doesn't grow exponentially in case some devices stop responding and the user continues to try and send messages to it. Entries will eventually time out, and make space for new entries.

Note that the application has only received a successful *confirmation message* so far, implying that the server has successfully received this command. Eventually when it receives the ZIGBEE_GENERIC_RSP_IND message (Z-Stack Linux Gateway API document, Section 7.1.2), the function *sr_process_generic_response_indication()* (in the same file) tries to match the received sequence number against all entries in this table. On finding a match, the application could take further action to update the display. Saving the sequence number on receipt of the confirmation message allows the application to correlate the received response with the command it sent out.

Note: In the sample application, the display isn't updated immediately after the receipt of the ZIGBEE_GENERIC_RSP_IND. In fact, on finding a match for the sequence number, another request is sent to read the attributes of the device in question. The display is updated only when a response to this "attribute read" is received. The reason for this extra iteration of commands is that we want the UI of our application to reflect the *actual* value of attributes of the device, and not blindly update it with the value that the user asked it to update to. A study of the source code in the abovementioned file might make this part clearer.

7.7 Miscellaneous considerations for Gateway Applications

This section goes through some application specific considerations that might be useful to keep in mind when implementing your own Gateway application.

- Gateway application should always try to reconnect to the servers as soon as it detects a disconnection. As soon as connection is reestablished to the Network manager it will be sent a Reset confirmation message NWK_ZIGBEE_SYSTEM_RESET_CNF

- The list of APIs supported by the various servers is substantial, but it doesn't cover all the ZCL APIs supported by ZigBee. However, the application can always send a raw ZCL frame over the air, using this API GW_SEND_ZCL_FRAME_REQ. Similarly when the Gateway subsystem receives an unprocessed ZCL frame, it sends it to the application using the following API GW_ZCL_FRAME_RECEIVE_IND.

- If your Gateway application needs to poll a particular device for its attributes, it should configure reporting using these APIs: GW_SET_ATTRIBUTE_REPORTING_REQ, GW_SET_ATTRIBUTE_REPORTING_RSP_IND, GW_ATTRIBUTE_REPORTING_IND. This is useful for instance, if the application wants to report values from a temperature or humidity sensor every few minutes or so. The Gateway sample application includes an example of attribute reporting configuration for temperature/humidity/occupancy sensors (see *<GATEWAY SAMPLE APP>/demo/framework/user_interface.c: ui_send_sensor_read()*).

- As mentioned in Section 6.5, NWK_GET_DEVICE_LIST_REQ API can be called by the Gateway application at startup to obtain a list of devices and their information that has been saved in the device database. However, the entries in this database may have become stale since the last run, so the application should ideally call NWK_DEVICE_LIST_MAINTENANCE_REQ API to clean up the device status returned by the first API.

8. Power and Reset Commands

The *zigbeeHAgw* script is responsible for bringing up all the servers in the correct order in order to setup and create the ZigBee network. It also tracks all the servers and if any server other than the Network Manager goes down, it will kill all the servers and bring them up again in the correct order. The network and device databases remains intact, so it retains memory of all previously-added devices and their addresses etc. The demise of the network manager is handled differently.

The Network manager supports 5 exit codes: OFF, RESET_HARD, RESET_SOFT, SLEEP, WAKEUP. Whenever a power down (NWK_SET_ZIGBEE_POWER_MODE_REQ) or Reset (NWK_ZIGBEE_SYSTEM_RESET_REQ) command is sent to the Network Manager, it exits with one of the above codes after putting the ZigBee device in appropriate low-power state. The POWER and RESET functionality of the Network manager is managed partially by the *zigbeeHAgw* script. The script keeps track of the exit code with which the Network manager exits, and acts accordingly.

- In all cases, it first stops all the servers
- If the exit code is OFF, that means the intention was to wind down the system (N NWK_ZIGBEE_SYSTEM_SELF_SHUTDOWN_REQ), so no more action is required.
- If Network manager exited due to a soft or hard reset or wakeup call, the servers are all restarted, but network manager is sent an command line argument "--reset_soft", "--reset_hard" or "--wakeup" so it can send appropriate confirmation message to the application after coming up.
- If the exit code is SLEEP, then the other servers are not brought up. Only the network manager is brought up with a command line argument "--sleep". This puts the network manager in a mode, where it only accepts the command to wake it up (NWK_SET_ZIGBEE_POWER_MODE_REQ), and doesn't process any other commands coming from the Gateway application. When the manager receives the wakeup command, it simply exits with a WAKEUP code, and lets the script handle the rest (see point above).

A lot of the above is implementation detail, but if the user intends to edit/add functionality to the *zigbeeHAgw* script, it is important to understand its role in correct operation of the servers. It also reiterates the importance of the application to continually maintain connection with the servers, since it may lose contact with them during the course of normal operation.

9. Configuration Files

Often a user might want to have more control over the network that is created by the servers, or over the local Gateway device itself. Configuration files are available that allow certain parameters to be tweaked to achieve this. Section 5 of the Z – Stack Linux Gateway User Guide lists all the configuration files distributed

with this release and the important parameters that it allows you to configure. Discussed below are some of the more common things that a user might want to tweak through these files.

9.1 ZigBee Network Parameters

A ZigBee network is created when the servers are run via the *zigbeeHAgw* script. This script individually brings up all the servers in the correct order, and also supplies them configuration files that they read upon startup. *config.ini* is the configuration file read by the Zstack ZNP server that provides the core ZigBee stack interface (Application Framework and ZigBee Device Object layers). This file contains parameters that let you configure the network created with the Z-Stack device. The user could change the channel the network uses via the parameter DEFAULT_CHANLIST, or it could provide a PAN ID that it wants the network to use via ZDAPP_CONFIG_PAN_ID. Proper Gateway operation requires the local Gateway device to act as the network Coordinator, so DEVICE_TYPE should be left as it (set to 0). If you would like the network to be open for joining devices on start up, you can tweak the parameter PERMIT_JOIN.

All the above parameters in *config.ini* are read at startup, and will be read-again if a hard reset is issued to the network manager via the NWK_ZIGBEE_SYSTEM_RESET_REQ command (Z-Stack Linux Gateway API document, section 8.1.1).

9.2 Defining End Points on the Gateway device

The local device that acts as the coordinator of the network is also a ZigBee node and can support ZigBee endpoints and clusters. This allows the local device to 'act' like a ZigBee device and communicate with the endpoints on other devices in the network. All messages sent by other devices to this endpoint would be received by the application.

An interesting use-case for defining endpoints would be if a ZigBee device in the network needed to control a non-ZigBee device. For instance, if we had a Wi-Fi Light that we wanted to control via a ZigBee switch in our network, we could represent that light via an endpoint defined on our local device. Binding the ZigBee switch with this local light endpoint, could allow the application to receive messages whenever the switch is operated. The application would typically receive this message as an incoming ZCL frame GW_ZCL_FRAME_RECEIVE_IND (Z-Stack Linux Gateway API document, Section 9.3.12) . This message could then be translated by the application into a Wi-Fi message that the switch understands.

The gateway endpoints can be defined in a configuration file called *gateway_config.tlg* that the Gateway server reads at startup. The file is re-read every time the application issues a reset via the NWK_ZIGBEE_SYSTEM_RESET_REQ command. Details of this configuration file, and an example of how to create some endpoints is provided in the Z-Stack Linux Gateway User Guide, Section 5.3.

The NWK_GET_LOCAL_DEVICE_INFO_REQ (Z-Stack Linux Gateway API document, Section 8.1.6) command can be used to query the gateway device for a list of registered endpoints and their descriptors. This API can be used by the application to display this information for the end-user is required. Z-Stack Linux Gateway API document, Section 9.3 has more Gateway APIs that can be leveraged to keep track of or bridge-over the attribute-values on local Gateway device endpoints.

The ability to define endpoints on the Gateway device is a very powerful feature that can be leveraged to create a host of interesting features in the User's Gateway application. The sample application also demonstrates this feature by allowing the user to record a single-command macro and associate with a gateway endpoint. This macro will now be recalled whenever a message is sent to the Gateway endpoint. Details of this feature, and instructions on using it, are in the Z-Stack Linux Gateway User Guide, Section 6.3.3.8.

