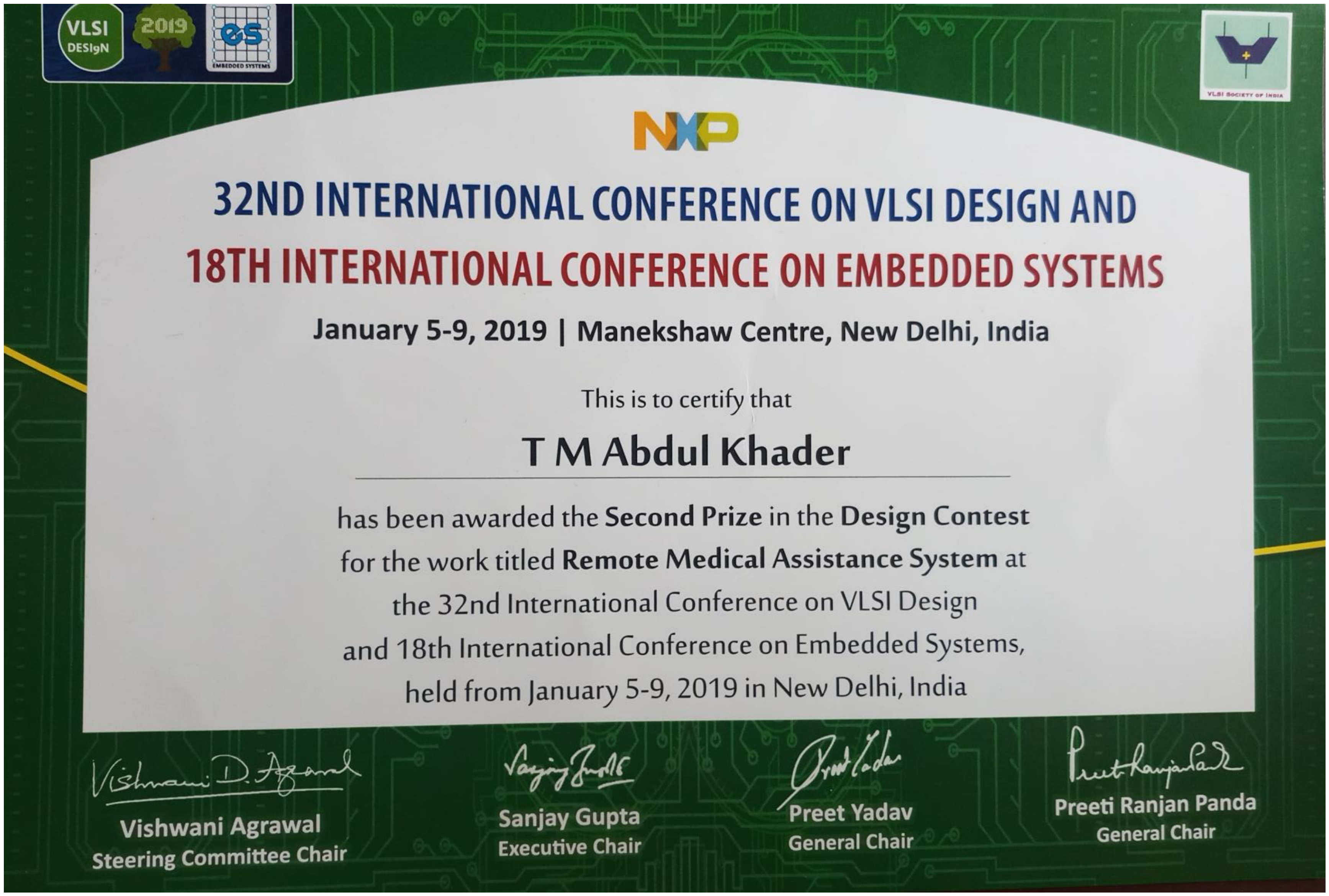
“Remote Medical Assistance System"

Implemented on NXP-Volansys Modular IoT Gateway

Report By:
T M Abdul Khader
IIT Guwahati



32ND INTERNATIONAL CONFERENCE ON VLSI DESIGN AND 18TH INTERNATIONAL CONFERENCE ON EMBEDDED SYSTEMS

January 5-9, 2019 | Manekshaw Centre, New Delhi, India

This is to certify that

T M Abdul Khader

has been awarded the **Second Prize** in the **Design Contest** for the work titled **Remote Medical Assistance System** at the 32nd International Conference on VLSI Design and 18th International Conference on Embedded Systems, held from January 5-9, 2019 in New Delhi, India

Vishwani D. Agrawal
Vishwani Agrawal
Steering Committee Chair

[illegible]
Sanjay Gupta
Executive Chair

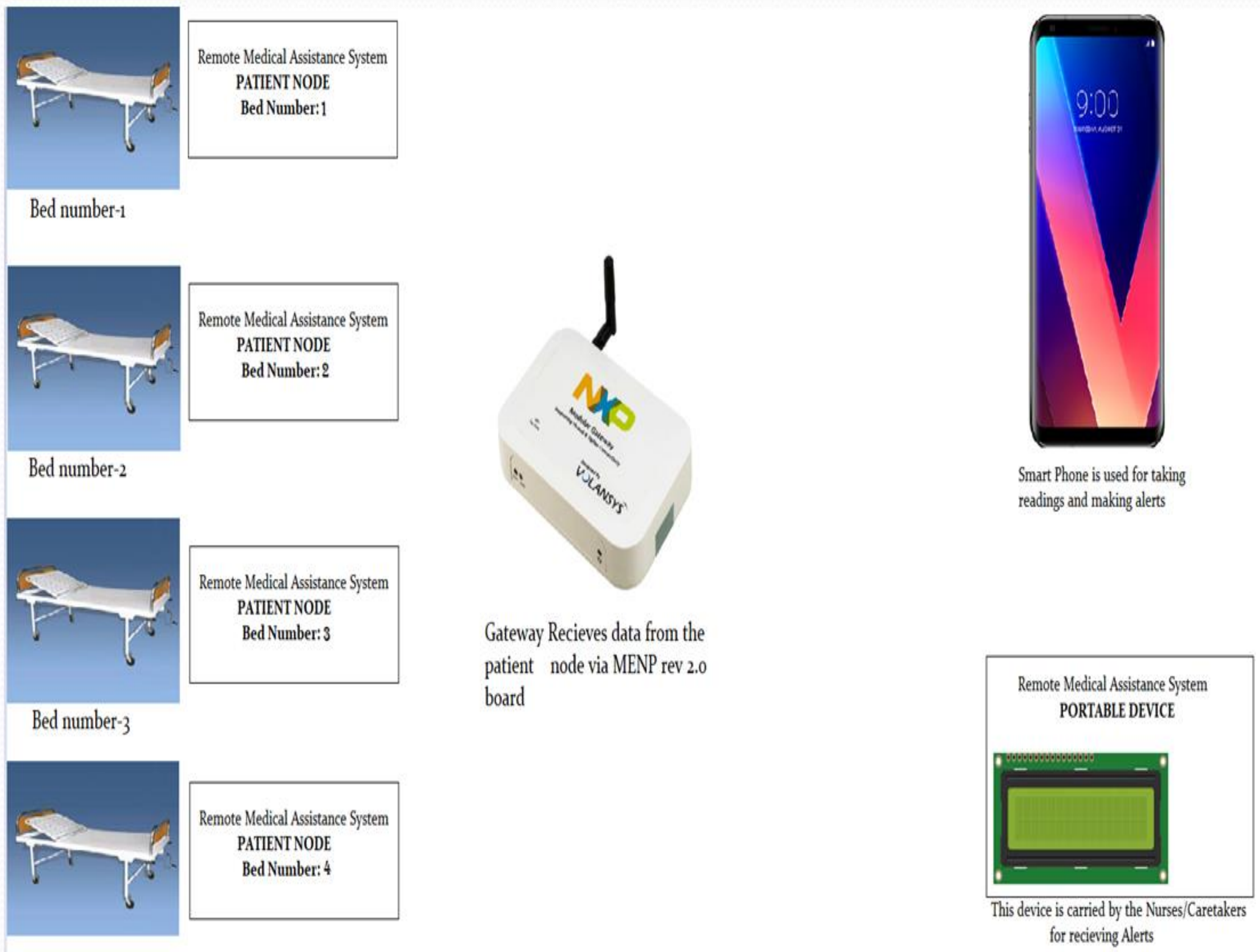
[illegible]
Preet Yadav
General Chair

[illegible]
Preeti Ranjan Panda
General Chair

Introduction: The architecture of this project was one among the selected 15 models among the ones presented by several Research Scholars, professors and Companies, which were selected for embedded implementation by NXP Semiconductors to be presented at The “32nd International Conference on VLSI Design and 18th International Conference on Embedded Systems”. For the implementation, we a team of two, Myself-T M Abdul Khader (B-Tech, IIT Guwahati) and Mr Sivakumar S (Research Scholar, IIT Guwahati) were provided with a kit constituting the NXP Modular Gateway, A microbus enabled MENP Node with a support of JN5179 wireless microcontroller which services as the end-node. The implementation took about a month with Mr Sivakumar handling the Android application and myself involving in the embedded development on the JN5179 controller. The model was presented at conference and turned out to be success which brought us 2nd prize with a cash award of Rs:30000 and lots of appreciation for our efforts. We were further invited to present our model at NXP Noida Company Campus.

Objective of the Project: Remote Medical Assistance system is intended to provide assistance to patients living in remote areas to have assistance from doctors situated anywhere in the world, the device lets doctor to have the vital readings of patient at any point of time. A single or a few caretakers can simultaneously handle several patients at a time in a clinic using their portable module and hence this forms a working model of an advanced Medical IoT prototype. The device could also be issued to patients whose condition is favorable enough such that they can treated at their domestic environments. In case of an emergency, doctor could voluntarily trigger an alarm for the caretaker so that he can immediately attend the patient. Automatic emergency detection is also enabled in the project.

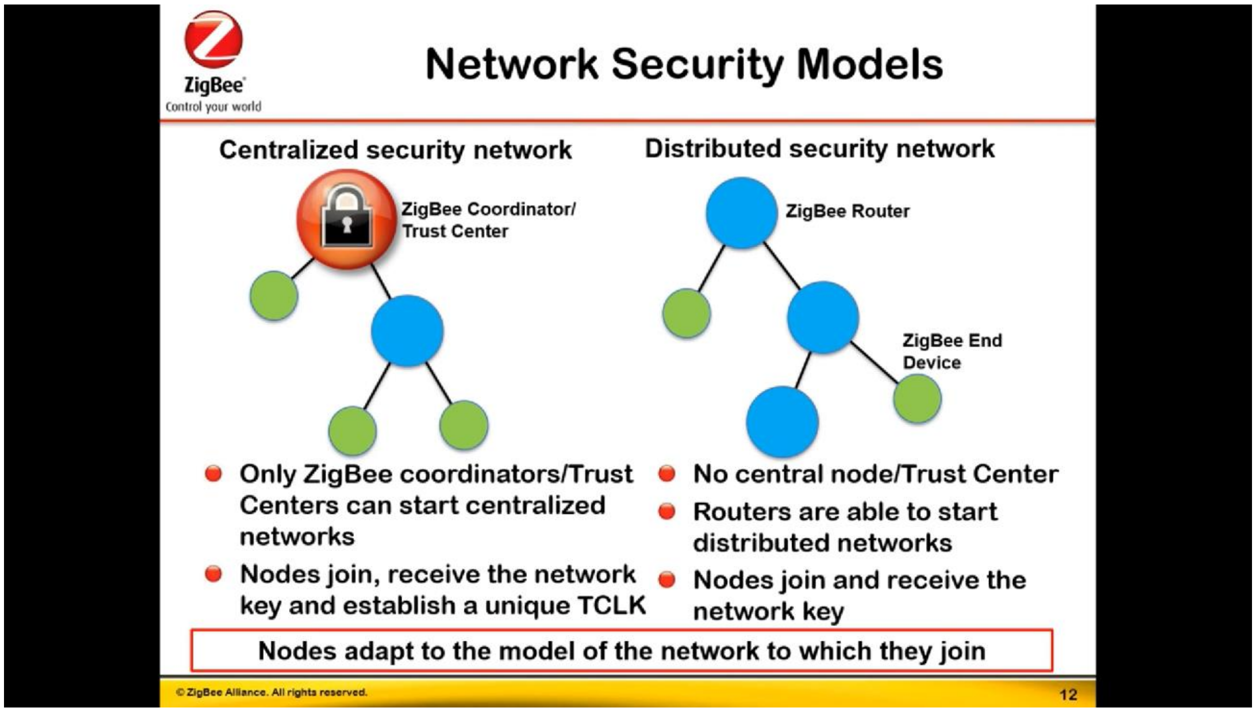
Future Prospective of Device: The data collected by the device could be processed in a centralized server so that it could be used to analyze various medical conditions. Analyzing the trend of these data at a clinic or an area could also give us hint about an out bursting medical situation such as a viral infection. With growing influence of machine learning and AI over medicine, we can even find out the medical trends and probability of a medical situation in a locality. However as a student we have a lot of limitations on such prospective but, teaming up with an efficient team can enable us to work for the same. Having several such prototypes all over the world connects the whole medical world so that we can put together the ideas of several fine minds of the world. Shared ideas and medical data can bring about cure to even the most deadly diseases we have currently to our disadvantage.


Remote Medical Assistance System
PATIENT NODE
Bed Number: 1
Bed number-1
Remote Medical Assistance System
PATIENT NODE
Bed Number: 2
Bed number-2
Remote Medical Assistance System
PATIENT NODE
Bed Number: 3
Bed number-3
Remote Medical Assistance System
PATIENT NODE
Bed Number: 4
Gateway Recieves data from the patient node via MENP rev 2.0 board
Smart Phone is used for taking readings and making alerts
Remote Medical Assistance System
PORTABLE DEVICE
This device is carried by the Nurses/Caretakers for recieving Alerts

Zig-Bee 3.0 An Introduction

Introduction: This module illustrates how to program JN5179 wireless micro-controller for zig-bee applications. It includes zigbee 3.0 network with Green power support.

Zigbee 3.0- Types of Connection…



Network Security Models

Centralized security network	Distributed security network
● Only ZigBee coordinators/Trust Centers can start centralized networks	● No central node/Trust Center
● Nodes join, receive the network key and establish a unique TCLK	● Routers are able to start distributed networks
	● Nodes join and receive the network key

Nodes adapt to the model of the network to which they join

ZigBee
Control your world
ZigBee Coordinator/ Trust Center
ZigBee Router
ZigBee End Device
© ZigBee Alliance. All rights reserved.
12

Process of establishing a connection in centralized network..

1)-Whenever a connection between a coordinator and node is required a user interaction at coordinator opens up the coordinator network for about 180 seconds

2)-When the coordinator network is open, a user interaction at the node prompts an association with the coordinator.

3)-Upon associating, network key and a unique TCLK (Trust centre link key is exchanged).

In Distributed security network any router is able to start a distributed network.

What can the nodes do…..

1) **Network Steering**- It can find and get on a network.

2) **Network formation**-It can itself form a network.

3)-**Touch-link**- Proximity commissioning

Network Steering- Procedures for node not on network…

1)-Upon a user interaction, it will first perform a channel scan and select one network from the available networks in case the device is not on any network.

2)-Once it choose a network, it will get a network key from the coordinator or a router

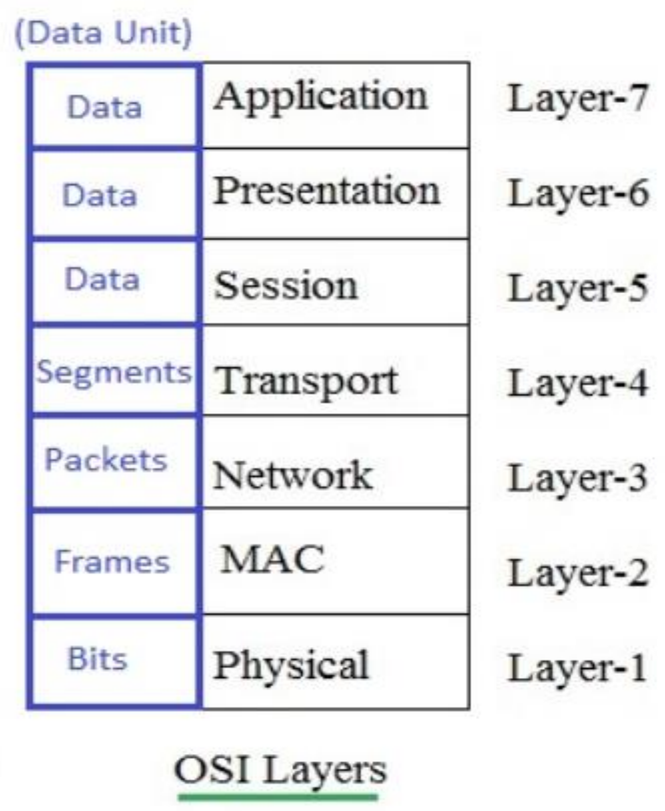
3)-If it's a centralized network a TCLK (Trust centre link key), a unique ID is exchanged between coordinator and node.

If on a network….

The network will be opened for about 180 seconds for other nodes to join..

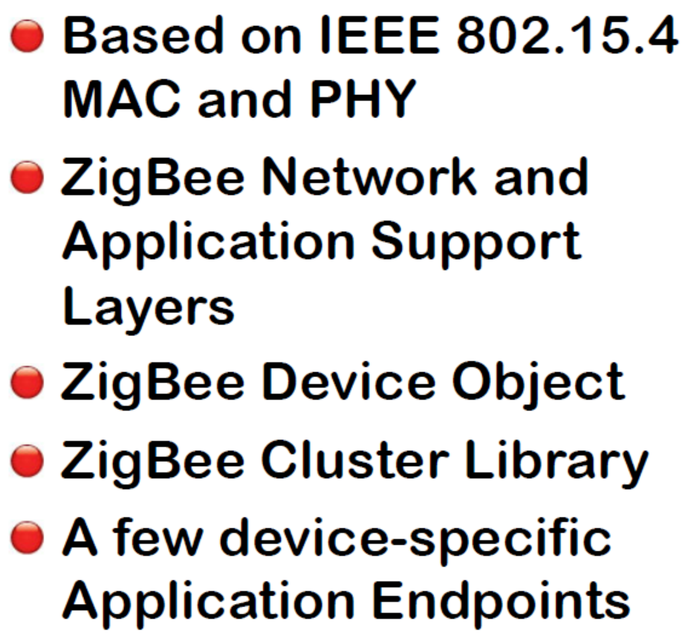
OSI (Open System Interconnection)

It is the conceptual approach of how any open network can establish a network and communicate over that network.

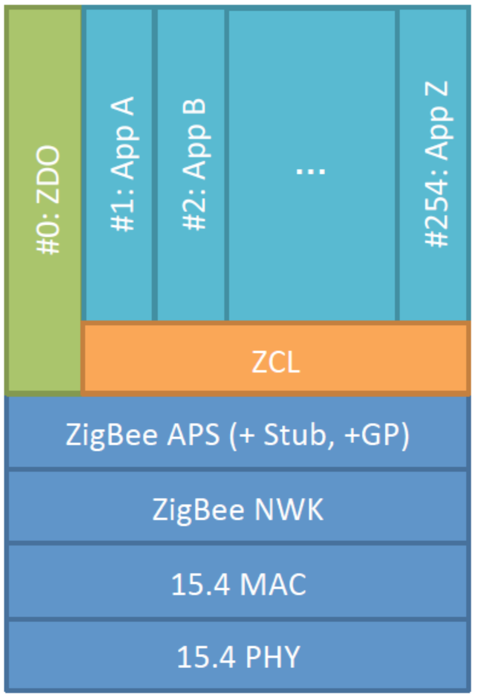


PHY (Physical layer): Transmission and reception of bit-streams over a physical medium
MAC (Media Access Control Layer): Transmission of data frames between two nodes connected by a physical layer
Network: Manages a multi-node network including addressing routing, traffic control
Transport: Reliable transmission of data segments between points on a network including segmentation, acknowledgement and multiplexing.
Session: Managing communication sessions. Ie handling continuous exchange of information in form of multiple back and forth transmission between two nodes
Presentation: Translation of data between a networking service and an application including character encoding data compression, encryption etc.
Application: High level API, resource sharing etc.

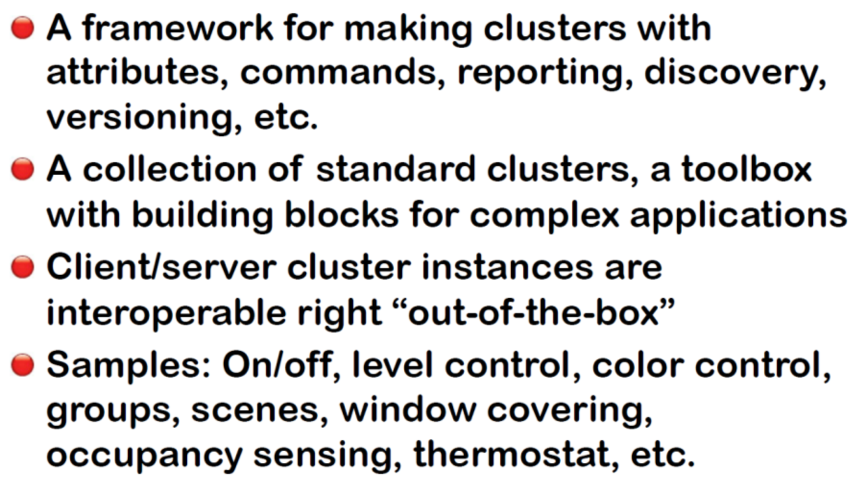
Networking in Zig-Bee 3.0 also works by the OSI Layers.



- Based on IEEE 802.15.4 MAC and PHY
- ZigBee Network and Application Support Layers
- ZigBee Device Object
- ZigBee Cluster Library
- A few device-specific Application Endpoints



In the OSI stack of Zigbee, ZCL stands for Zigbee-Cluster layers.
Zig-Bee clusters specify the functional domain of the system.
Example-On/Off clusters, level-control clusters etc…where on off cluster is used to switch on and switch off a device



- A framework for making clusters with attributes, commands, reporting, discovery, versioning, etc.
- A collection of standard clusters, a toolbox with building blocks for complex applications
- Client/server cluster instances are interoperable right "out-of-the-box"
- Samples: On/off, level control, color control, groups, scenes, window covering, occupancy sensing, thermostat, etc.

Zig-bee Network Nodes….

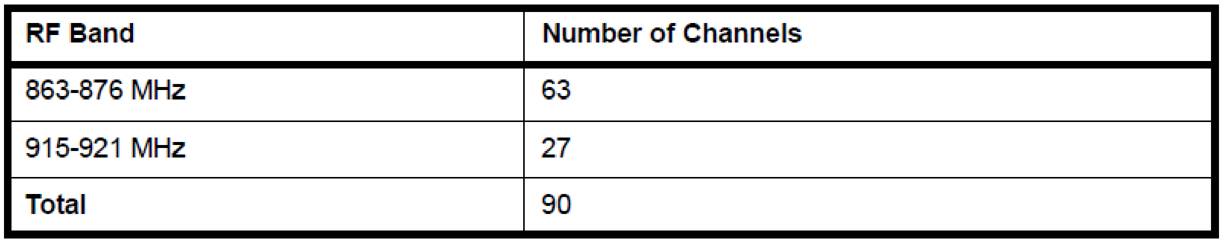
1)-**Coordinator**- Node that creates a network
2)-**Route**r-Router extends a network…it may itself process data.

3)-**End device**- The one which joins a given network. It cannot create a network

Topology

Zigbee pro engages mesh topology which constitute of several end nodes and router connected to a coordinator. Each end node has a parent which is either a coordinator or a router and an end endnode can only have a direct communication with its parent.

Frequency band distribution vs channels in a given band…



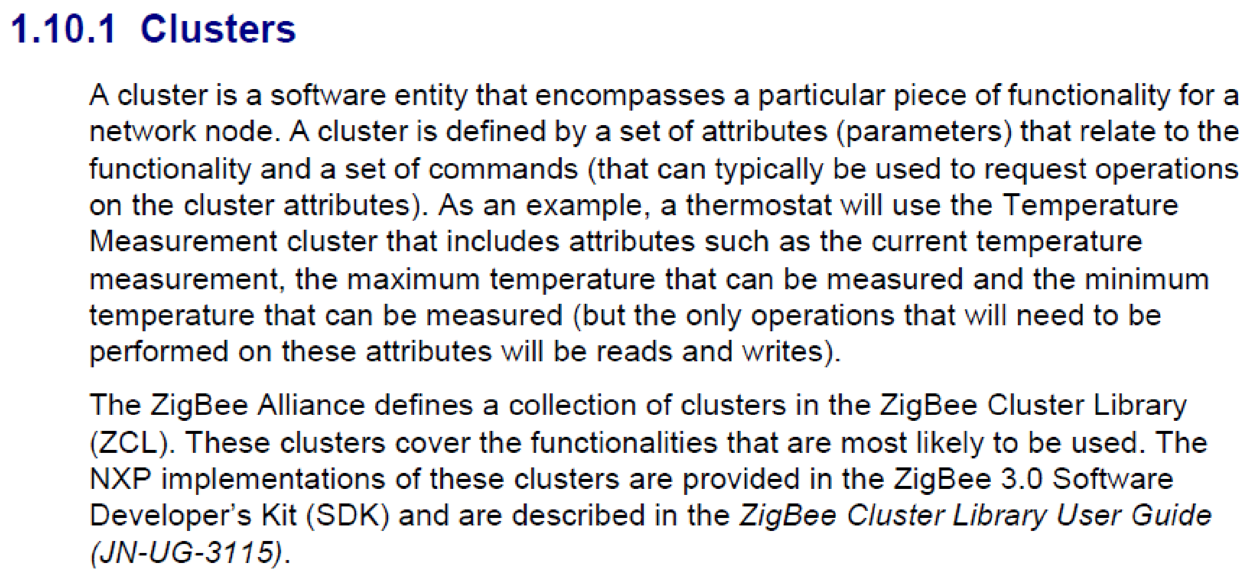
RF Band	Number of Channels
863-876 MHz	63
915-921 MHz	27
Total	90

The 2400 Mhz band is widely accepted. To represent an channel from the above we use a 32 bit channel mask. Note that to avoid noise, the most quite channel is always chosen to create the network.

Security

Bulk of security measures are taken to avoid noise in data as well as data stealing for avoiding noise techniques like Quadrature Phase shift keying, Listen before send algorithm etc are used. For security..128 bit AES based encryption system is used to avoid data stealing. This encryption is key based. Usually for a given network all nodes will be given the same network key. However another key called as TCLK is generated by a trust centre which can be any node (by default-coordinator). A TCLK is given to each node. Now any data transmission that occur in a given network will be encrypted by network key as well as TCLK.

Clusters

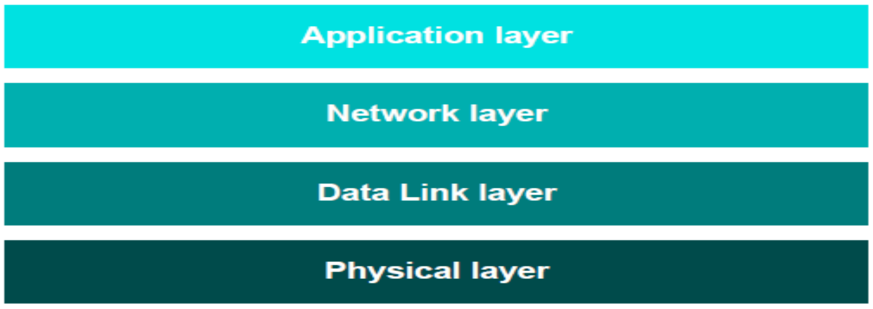


1.10.1 Clusters

A cluster is a software entity that encompasses a particular piece of functionality for a network node. A cluster is defined by a set of attributes (parameters) that relate to the functionality and a set of commands (that can typically be used to request operations on the cluster attributes). As an example, a thermostat will use the Temperature Measurement cluster that includes attributes such as the current temperature measurement, the maximum temperature that can be measured and the minimum temperature that can be measured (but the only operations that will need to be performed on these attributes will be reads and writes).

The ZigBee Alliance defines a collection of clusters in the ZigBee Cluster Library (ZCL). These clusters cover the functionalities that are most likely to be used. The NXP implementations of these clusters are provided in the ZigBee 3.0 Software Developer's Kit (SDK) and are described in the *ZigBee Cluster Library User Guide (JN-UG-3115)*.

Architectural Implementation



Physical Layer – Physical layer is created out of an IEEE standard for interchange of data bits with the medium here- radio and the progressive layers
Data Link Layer- I provides address to the given data ie where it comes from and where an outgoing data goes.
Network Layer- Provides the network functionality, in this case Zig-bee pro interface with the subsequent layers.
Application layer- This constitute of the basic application that is running on a given node which generates data to be transmitted.
This is the architecture for any sort of network implementation.

Network Addressing

MAC (IEEE) Address- It is a 64 bit address used to address a given node. It is unique that no two nodes in the world has the same mac address.
Network address: This address is a 16 bit address assigned to a node in a network. No two nodes in a network has same network address.

Network Identity

A given zig-bee pro network also should have unique identity to enable two networks to work in close proximity.
PAN ID: A 16 bit personal area network ID is used to identify a given network. This is generated randomly and shared among nodes in the network.
Extended PAN ID: It is the 64 bit version of PAN ID.

Starting a Network

1)-When a coordinator first start a network it will set an extended PAN ID and its own network address. This should be provided in the coordinator's program else the MAC ID of the coordinator will be chosen as the extended PAN ID.
2)-Next it runs an energy detection scan to choose the quietest channel in the band
3)-Next it will set the PAN ID for the network. This will be a random 16 bit number and the coordinator will also ensure that neighboring networks doesn't have the same PAN ID.
4)-Now it receives join requests from other nodes.

Joining a Network

1)- The end node first performs a run across channels in the given RF band to search for networks. There might be multiple networks in a single channel, the required network is chosen by pre-fed extended PAN ID of the required network.
2)-After choosing the network, in the mesh the parent with the least distance from coordinator is chosen.
3)-A request is sent by the node to join the network.
4)-Once the parent receives a request it will consult with the Trust centre and either lets or denies the node to enter the network.
5)- After joining the network, the node learns PAN ID , extended PAN ID as well as its network key. This enables it to rejoin in future in case it breaks away from the network.
Attributes…
A data structure handled by Zig-bee is called as an attribute. An example is temperature in case of temperature sensor.
Clusters use these attributes to perform functionalities. Ie clusters
ZCL
To adhere with the interoperability of zig-bee with other networks developers have developed several standard cluster library called ZCL (Zig-bee cluster library) for users or developers to refer while creating a network application.

ZIG-BEE STACK SOFTWARE FOR JN51xx

The zig-bee 3.0 software stack for NXP enables smooth functionality of zig-bee 3.0 using additional C libraries and software development kits.
The Zigbee pro API
There are three types of zig-bee pro APIs
1)-ZDO API-Zigbee Device objects is concerned with management of local device…say adding a device into network
2)-ZDP-Zigbee device profile
3)-AF-Concerned with creating data frames for transmitting and modifying device descriptors.

Functional Explanation of the Code

main.c- Initiates software timers defined in ZTimers.h depending upon the functioning modes say- Green power mode, NFC reading timers and OTA update delay timers. The main application loop defined as “APP_vmainLoop(void)” involves initializing the zigbee pro stack tasks defined by zps_taskZPS(). This is an exported function meaning that it is declared globally across the source files that it could be used elsewhere. Then it defines the Zigbee Base Device behavior task followed by starting the timer tasks. This is followed by the main execution part, the APP_tasklight(), which carries out the task of receiving a zig-bee cluster event, recognizing the type of event and then carrying out necessary function at the endpoints. Once the task is accomplished, the vAHI_watchdog_restart() restarts the watchdog which indicates that the essential task of the node in the given cycle is over.

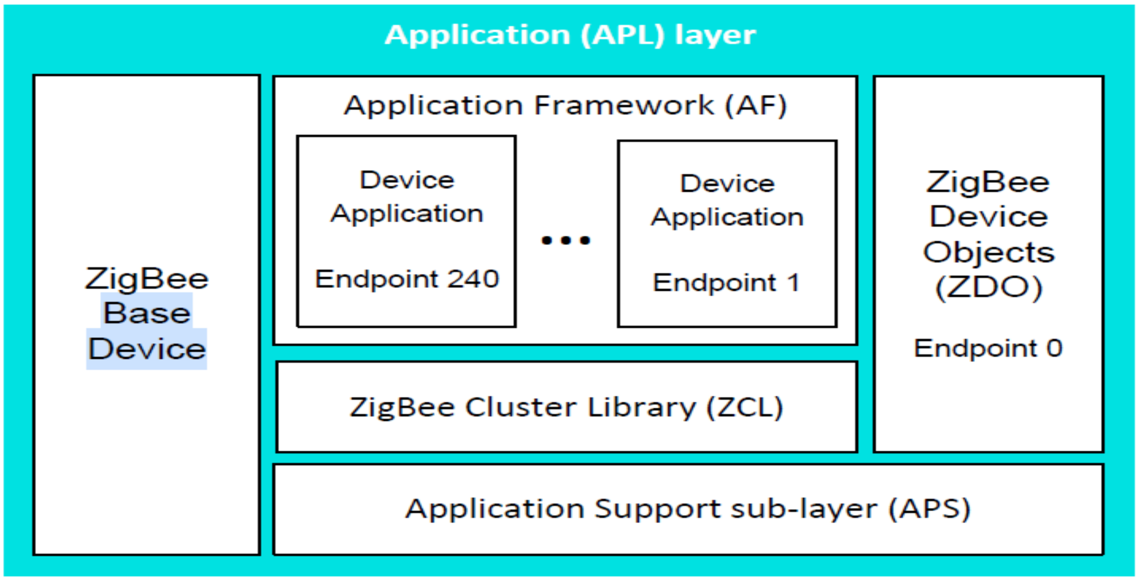
app_zlo_light_node.c: This c file contains the function APP_tasklight() which is directly called from the main loop and this is the primary edit in the project. Each time it is called it checks an activity in a specific event queue, say in our case, the button press event, and using a library defined function, `ZQ_bQueueReceive`, we receive the message to a specific pointer address of a structure. Now we extract different features of message one of which is event type and check whether it is a button up or button down event. Note that this is the primary edit created in the program that in case of a button press a software timer initialized in main.c is called and started, the name of counter is “counter” and called as `ZTIMER_eStart(counter, ZTIMER_TIME_SEC(10000))` This counter is called of in case of a button down event and the final count is extracted and decoded to get the intended value of reading. Once the value is decoded, it is send to the required device using the function `vSendButtonCount()`.The definition of this function is also explained in the same file (Please refer the same). The message is sent inform of a structure which defines the destination address of the node where the data is to be received, the endpoint where the data has to be send. It used a predefined structure in zcl whose declaration is, `tsZCL_Address    psDestinationAddress;` where all the required data like group address, 16 bit destination address,64 bit destination address, broadcast mode etc is specified. Now the above details specify the address of destination node now data is send using another function defined in same file called, `vSendCustomData` which takes the decoded data and depending upon the address mode, using a switch statement, the data is transmitted by specifying the endpoint. The rest of the source code file is mainly concerned with zig-bee device object functions such as joining and leaving a network which is not of much significance to the purpose of project.

app_buttonhandler.c: The main function involved initializing the pin in which button connected as input. Note that in our project we are not using the button explicitly but controls the logic levels of the connected pin. Input pull up is enabled so the given pin is active low (to avoid noise effects). The selected pin is interrupt enabled and active edge is also configured so that whenever there is a change in logic levels, the system is notified. The interrupt service routine is defined under function, `vISR_SystemController` which is defined in this c file, while the ISR or registered in main.c by`vAHI_SysCtrlRegisterCallback ( vISR_SystemController )` , The interrupt service routine checks whether the given 32 bitmap matches with the DIO mask, for the interrupts then disables the interrupt until scan is complete and calls a timer function to which the button task function is passed as call back `ZTIMER_eOpen(&u8TimerButtonScan, APP_cbTimerButtonScan, NULL, ZTIMER_FLAG_PREVENT_SLEEP);` The call back function which is the button scan function implements a debounce logic to ensure a proper event by checking the state for 8 scan periods, see what type of event it is, ie whether a button up or button down event, and sends the message to `APP_msgAppEvents` ,queue using the function defined in ZCL as `ZQ_bQueueSend(&APP_msgAppEvents, &sButtonEvent)`, Now note the function in the previous c file which regularly checks for the given queue for any events and takes the necessary actions. Now the c files and codes until now explains how the data from sensor is send to gateway by JN5179.

app_RelayClick.c :The c file which controls the hardware output action that controls relay click which we have manipulated in this project. Instead of taking the relay click output, we directly take logic state output from the DIO ports to control Arduino to take sensor readings. The DIO pin states is indicated by a 32 bit bitmap which can be found in the JN5179 documentation and in the program a mask is defined for the bitmap mask of relay pin1 and relay pin2 as `DIO_PIN_RELAY_CLICK_2_MASK`, `DIO_PIN_RELAY_CLICK_1_MASK`,and the macro expansion states that Mask 2 is 1 left shifted by 6 times and Mask 1 is 1 left shifted by 13 times now using the bitmap definition, we can easily find out the DIO pins corresponding to the relay pins.There are two functions defined for activating DIO pins and deactivating them with input parameters as relay ID which is used to set the corresponding pins as logic 1 or logic 0, using the command `vAHI_DioSetOutput(u32Outputs, 0)`, where u32Outputs is the corresponding BITMAP of the relay ID.

App_OnOffLight.c :This c file stands as an intermediate in controlling the state of relays using the previous c file RelayClick.c by handling the ZCL events received from the ControlBridge. The relay modules or their DIO pins are controlled by the On/Off cluster defined in zcl_options.h as “CLD_ONOFF” it is defined as the ZCL server so that it can receive messages and control switching ON and switching OFF the DIO pins. The file begins by defining device table for both relays by setting them up at two different endpoints. This is followed by a function `eApp_ZLO_RegisterEndpoint`,a function which takes in a function pointer as parameter. This function once called registers the address of both endpoints and once it is done, the callback function is called using the function pointer. This c file also contain a function for ZLO device initialization. `sLight.sOnOffServerCluster.bOnOff = TRUE`, This is the command used for setting relay one or DIO pin for relay 1 ON by default which can be find in the same c file. You can also decide which endpoint to receive ZCL messages using the function **`bAllowInterPanEp`**, Until now whatever discussed is how a ZCL event is analyzed to take an action that controls the state of DIO pins connected to relay modules. Now we have to understand about how a ZCL event is received in queue.

app_zcl_light_task.c: **`APP_ZCL_vInitialise`**,is used to initialize the ZCL related functions with a callback function pointer `APP_ZCL_cbGeneralCallback`, as one of the parameters. It registers the endpoints using the function defined in previous c file and makes device specific initializations like turning relay one 1 by default, once connected. It also kick starts a timer `ZTIMER_eStart(u8TimerTick, ZCL_TICK_TIME)`,which has a callback function associated with it which is defined in the same c file as **`APP_cbTimerZclTick`** , which creates the zcl call back event structure as `tsZCL_CallBackEvent sCallBackEvent`, sCallBackEvent is passed to ZCL event handler `vZCL_EventHandler(&sCallBackEvent)`, another of the most important function in this c file is **`APP_ZCL_vEventHandler`**, THIS IS THE KEY FUNCTION which takes care of a ZCL task and is called from a function **`APP_vBdbCallback`**, in the zlo.c file now this function is called by the main loop function **`bdb_taskBDB`**,whose definition if you see you can see a statement as `ZQ_bQueueReceive(sBDB.hBdbEventsMsgQ, &sZpsAfEvent)`, This statement return TRUE if the queue receives a BDB event which consists internally information about the cluster event. So basically in main loop, the program constantly checks for a BDB event and if the BDB event is ZCL event, the previous functions are synchronously triggered finally reaching upto the activation or deactivation of the DIO pins.


Application (APL) layer
ZigBee Base Device
Application Framework (AF)
Device Application
Endpoint 240
...
Device Application
Endpoint 1
ZigBee Device Objects (ZDO)
Endpoint 0
ZigBee Cluster Library (ZCL)
Application Support sub-layer (APS)

Elaborated Working of Functions in the Embedded Program

Setting up of network by coordinator

1)-In this project we use Zig-Bee 3.0 IoT control bridge to Application to setup trust centre and form a network. Now once the coordinator is running (Jn5179 control bridge), We have to do network steering in order to prompt nodes to join the network.

- On a Control Bridge, configure the device as a Router, set up the channel mask and run the E_FL_MSG_START_SCAN (0x25) command.

The above command is used to scan for nodes.

<u>End-Node Relay Click Code explanation</u>

app_RelayClick.c

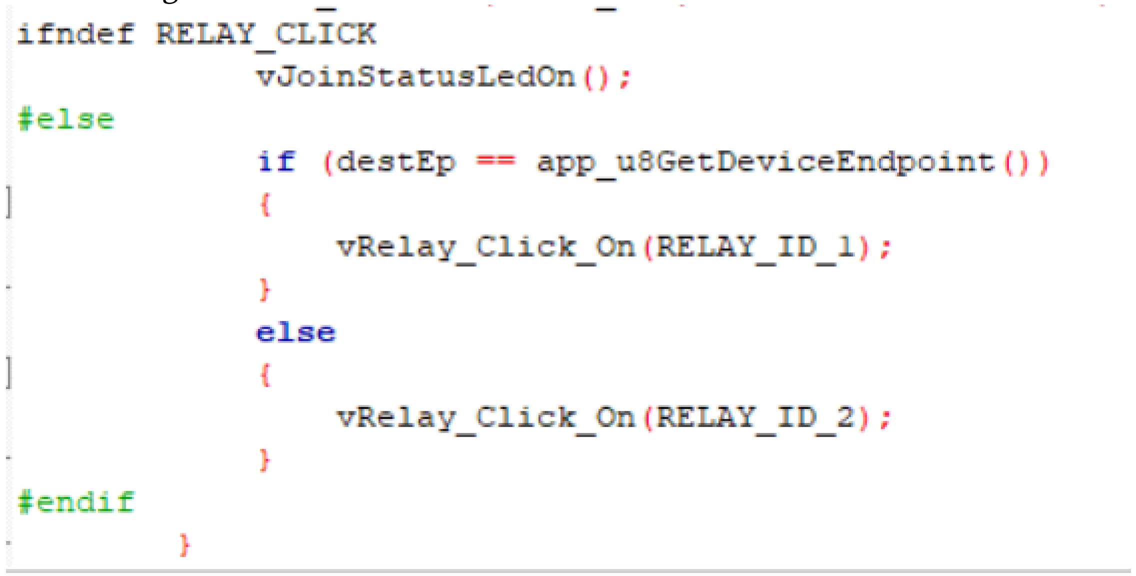
```
void vRelay_Click_Init (void)
{
   uint32 u32Outputs = (DIO_PIN_RELAY_CLICK_2_MASK |
DIO_PIN_RELAY_CLICK_1_MASK);

   vAHI_DioSetDirection (0, u32Outputs);
}
```

The vAHI_DioSetDitection command is used to set port type of pin as input or output The first argument sets the corresponding bitmap of pin as output by changing one of the bits. The second argument sets the pin as input. Here u32_output is the bit map of the pin of relay1 or relay 2. So by the given command relay pin will be set as output.

vRelay_Click_On()

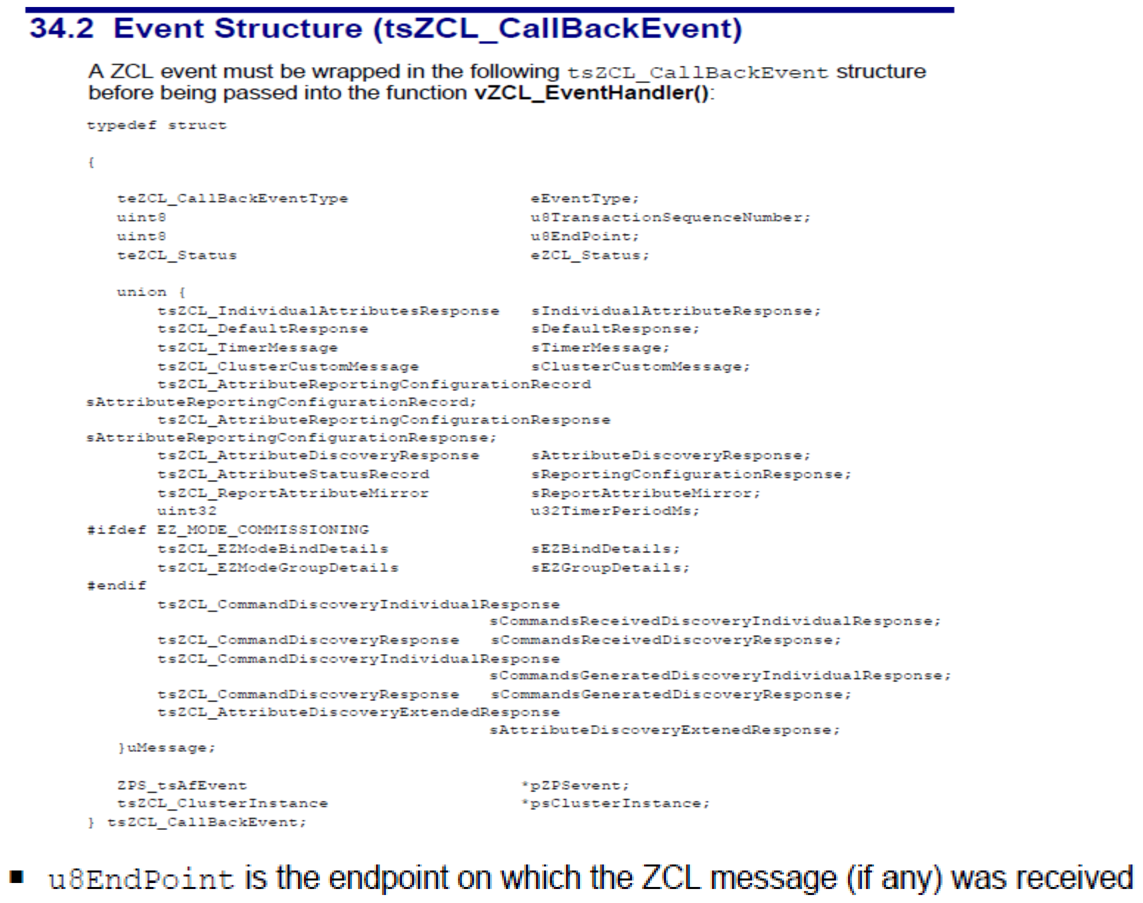
```
void vRelay_Click_On (relayed, relayNo)
{
   uint32 u32Outputs;

   DBG_vPrintf(TRACE_REALY_CLICK, "\nRelay Click %u On\n", relayNo);

   if (relayNo == RELAY_ID_1)
   {
      u32Outputs = DIO_PIN_RELAY_CLICK_1_MASK;
   }
   else
   {
      u32Outputs = DIO_PIN_RELAY_CLICK_2_MASK;
   }

   /* Set GPIO High. */
   vAHI_DioSetOutput(u32Outputs, 0);
}
```

The DBG_vprintf() will print the given string ie Relay click followed by relay number if the Boolean literal given as the first parameter is true.
When the relay_clickOn() is called the above string will be printer through UART.
Now depending on the value of relay ID, one of relay_pin_1_mask or relay_pin2_mask will be selected as the output pin u32output.
vAHI_DioSetOutput() has two parameters giving the pin at the first parameter will the pin active and giving at second parameter will deactivate it.
PRIVATE void vhandleOnOffFromZCL(uint8_t destEp)
(app_zcl_light_task.c)
The above function is called to invoke relay_clickOn() and relay_clickOff() to set the relay pins ON and OFF.
The thread given below controls it...



```
ifndef RELAY_CLICK
            vJoinStatusLedOn();
#else
            if (destEp == app_u8GetDeviceEndpoint())
            {
                vRelay_Click_On(RELAY_ID_1);
            }
            else
            {
                vRelay_Click_On(RELAY_ID_2);
            }
#endif
        }
```

destEP is an 8 bit data passed to the function whose value compared with the output of _u8GetDeviceEndpoint() decides whether the relay pin should be turned ON or OFF
vhandleOnOffFromZCL(psEvent->u8EndPoint);
Now psEvent is an identifier of structure datatype tsZCL_CallBackEvent
Defenition of the structure-



34.2 Event Structure (tsZCL_CallBackEvent)

A ZCL event must be wrapped in the following `tsZCL_CallBackEvent` structure before being passed into the function **vZCL_EventHandler()**:

```
typedef struct

{

   teZCL_CallBackEventType                     eEventType;
   uint8                                       u8TransactionSequenceNumber;
   uint8                                       u8EndPoint;
   teZCL_Status                                eZCL_Status;

   union {
        tsZCL_IndividualAttributesResponse    sIndividualAttributeResponse;
        tsZCL_DefaultResponse                  sDefaultResponse;
        tsZCL_TimerMessage                     sTimerMessage;
        tsZCL_ClusterCustomMessage             sClusterCustomMessage;
        tsZCL_AttributeReportingConfigurationRecord
sAttributeReportingConfigurationRecord;
        tsZCL_AttributeReportingConfigurationResponse
sAttributeReportingConfigurationResponse;
        tsZCL_AttributeDiscoveryResponse       sAttributeDiscoveryResponse;
        tsZCL_AttributeStatusRecord            sReportingConfigurationResponse;
        tsZCL_ReportAttributeMirror            sReportAttributeMirror;
        uint32                                 u32TimerPeriodMs;
#ifdef EZ_MODE_COMMISSIONING
        tsZCL_EZModeBindDetails                sEZBindDetails;
        tsZCL_EZModeGroupDetails               sEZGroupDetails;
#endif
        tsZCL_CommandDiscoveryIndividualResponse
                                          sCommandsReceivedDiscoveryIndividualResponse;
        tsZCL_CommandDiscoveryResponse    sCommandsReceivedDiscoveryResponse;
        tsZCL_CommandDiscoveryIndividualResponse
                                          sCommandsGeneratedDiscoveryIndividualResponse;
        tsZCL_CommandDiscoveryResponse    sCommandsGeneratedDiscoveryResponse;
        tsZCL_AttributeDiscoveryExtendedResponse
                                          sAttributeDiscoveryExtenedResponse;
   }uMessage;

   ZPS_tsAfEvent                              *pZPSevent;
   tsZCL_ClusterInstance                      *psClusterInstance;
} tsZCL_CallBackEvent;
```

- `u8EndPoint` is the endpoint on which the ZCL message (if any) was received

So as per the the function destEP is the destination endpoint on which ZCL message was received.

app_start_light()

(App_OnOffLight.c)

PUBLIC uint8 app_u8GetDeviceEndpoint(void)

{

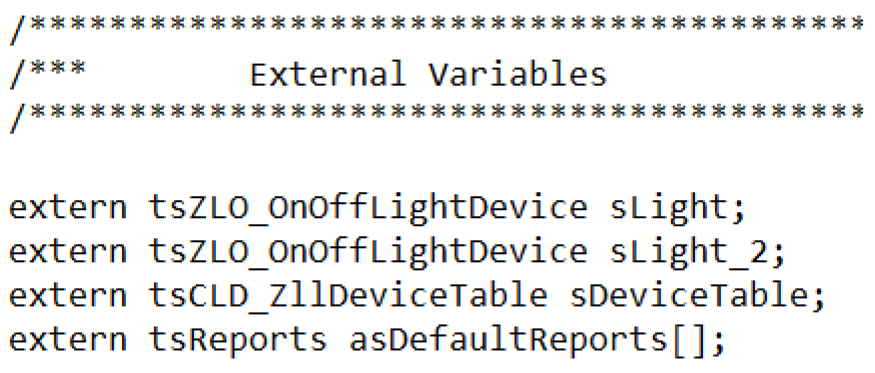
   return ONOFFLIGHT_LIGHT_1_ENDPOINT;

}

Now ONOFF_LIGHT_1_ENDPOINT is one of the daughter identifier of identifier *sDeviceTable* of structure data type tsCLD_ZllDeviceTable.

Now intuitively we can understand from the definition of sDeviceTable that ONOFF_LIGHT_1ENDPOINT is the endpoint address of relay 1.

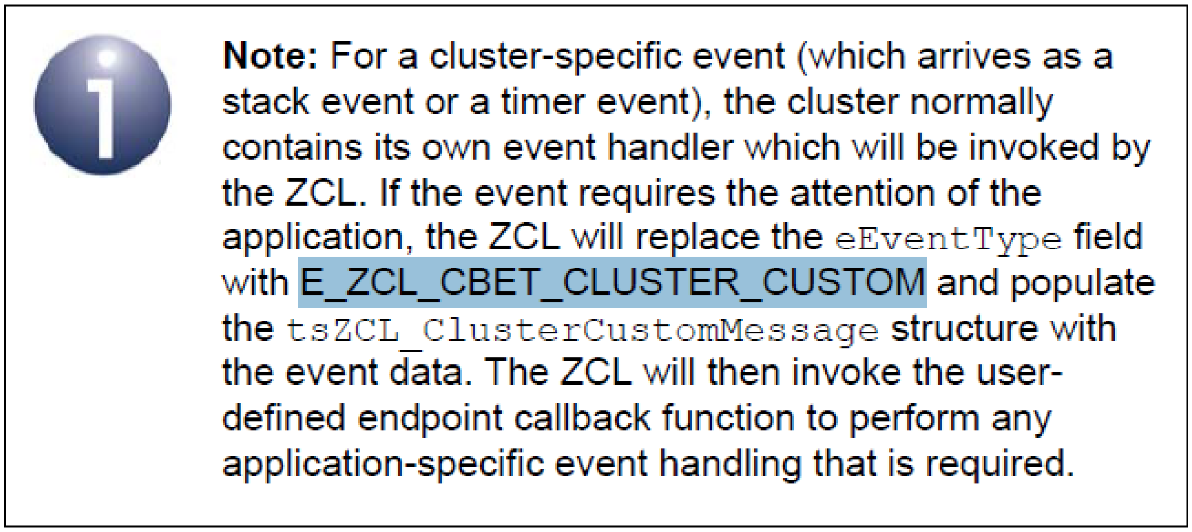
Now seeing the overall definition we understand that if the endpoint on which ZCL message was received is same as the endpoint of relay 1, switch relay one ON. Else switch relay 2 ON. Now we can see in the definition of sDeviceTable that only two endpoints are defined. These two endpoints stand for the two MENP relay modules.

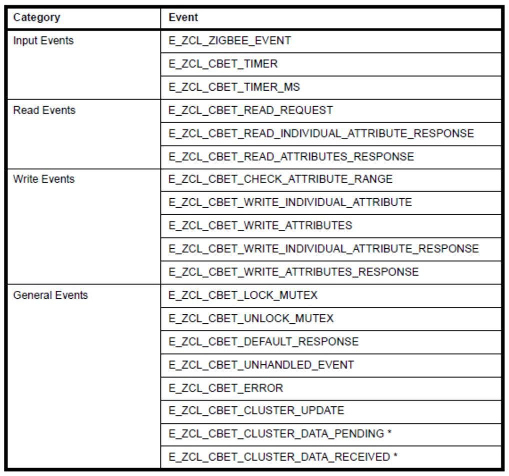


```
/******************************************
/***          External Variables
/******************************************

extern tsZLO_OnOffLightDevice sLight;
extern tsZLO_OnOffLightDevice sLight_2;
extern tsCLD_ZllDeviceTable sDeviceTable;
extern tsReports asDefaultReports[];
```

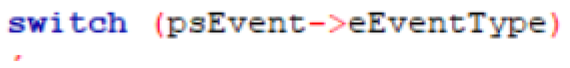
Now as of App_OnOffLight.c, sDevice Table is an exported variable

Note: Number of endpoints is defined in zcl_options.h.
Now refer app_zcl_light_task.c and function
PUBLIC void* psGetDeviceTable(void)
Now our next question is who handles the ***vhandleOnOffFromZCL(uint8_t destEp)*** function so that the destination endpoint is compared with the endpoint definition in device table to switch on the corresponding relay.
It is called by
APP_ZCL_cbEndpointCallback(tsZCL_CallBackEvent *psEvent)
Which is an endpoint specific callback for a ZCL event, where the details of present state event is passed by pointer.
We have to now find under what conditions the above function calls for vhandleOnOffFromZCL()
Under stand this


Note: For a cluster-specific event (which arrives as a stack event or a timer event), the cluster normally contains its own event handler which will be invoked by the ZCL. If the event requires the attention of the application, the ZCL will replace the `eEventType` field with E_ZCL_CBET_CLUSTER_CUSTOM and populate the `tsZCL_ClusterCustomMessage` structure with the event data. The ZCL will then invoke the user-defined endpoint callback function to perform any application-specific event handling that is required.

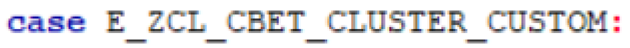

Category	Event
Input Events	E_ZCL_ZIGBEE_EVENT
	E_ZCL_CBET_TIMER
	E_ZCL_CBET_TIMER_MS
Read Events	E_ZCL_CBET_READ_REQUEST
	E_ZCL_CBET_READ_INDIVIDUAL_ATTRIBUTE_RESPONSE
	E_ZCL_CBET_READ_ATTRIBUTES_RESPONSE
Write Events	E_ZCL_CBET_CHECK_ATTRIBUTE_RANGE
	E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE
	E_ZCL_CBET_WRITE_ATTRIBUTES
	E_ZCL_CBET_WRITE_INDIVIDUAL_ATTRIBUTE_RESPONSE
	E_ZCL_CBET_WRITE_ATTRIBUTES_RESPONSE
General Events	E_ZCL_CBET_LOCK_MUTEX
	E_ZCL_CBET_UNLOCK_MUTEX
	E_ZCL_CBET_DEFAULT_RESPONSE
	E_ZCL_CBET_UNHANDLED_EVENT
	E_ZCL_CBET_ERROR
	E_ZCL_CBET_CLUSTER_UPDATE
	E_ZCL_CBET_CLUSTER_DATA_PENDING *
	E_ZCL_CBET_CLUSTER_DATA_RECEIVED *

The above table indicates the types of general events possible independent of type of cluster.



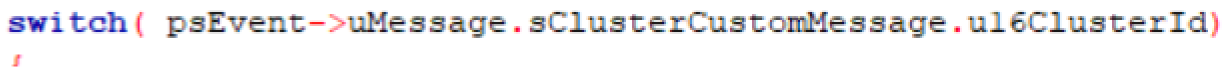
```
switch (psEvent->eEventType)
```

Once the endpoint callback() is called it checks for the event type. If the event type is E_ZCL_CBET_CLUSTER_CUSTOR. It means that the event is a customized cluster specific event example:ON/OFF event, like the one we handle here.



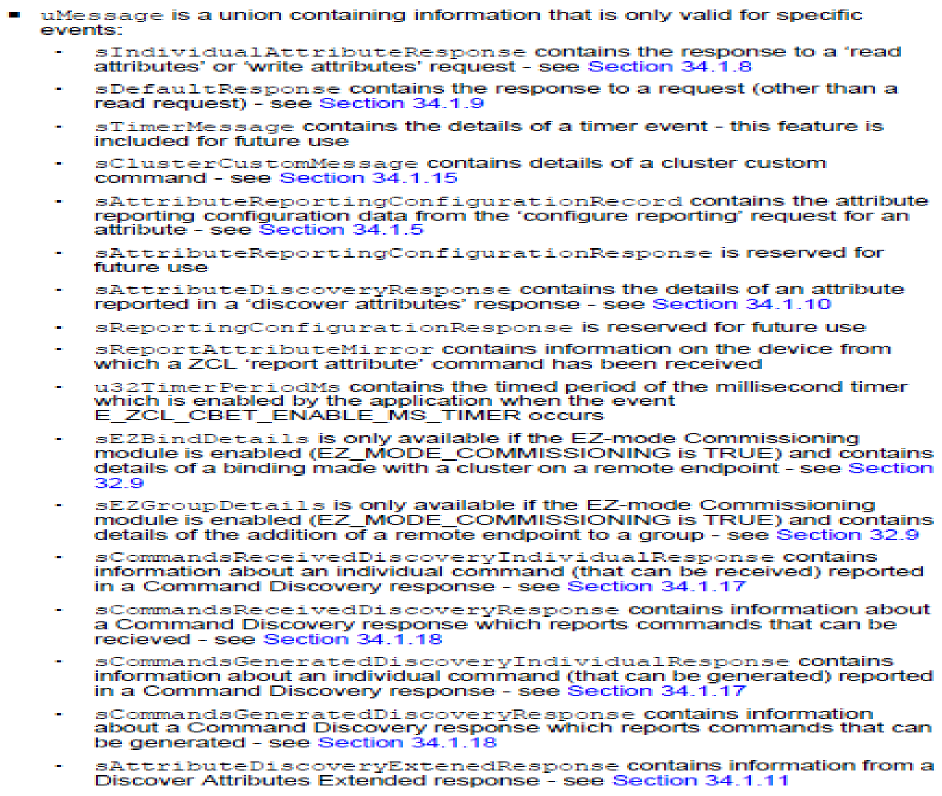
```
case E_ZCL_CBET_CLUSTER_CUSTOM:
```

If this case is found true,....



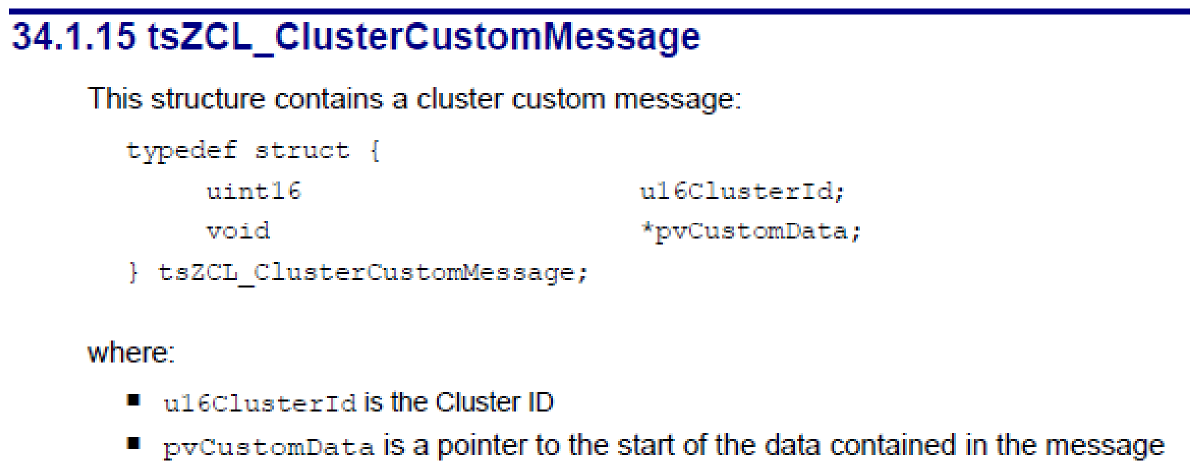
```
switch( psEvent->uMessage.sClusterCustomMessage.u16ClusterId)
```

Is another switch statement which is triggered. Now refer the table below for the possible outcomes of uMessage.



- `uMessage` is a union containing information that is only valid for specific events:
  - `sIndividualAttributeResponse` contains the response to a 'read attributes' or 'write attributes' request - see Section 34.1.8
  - `sDefaultResponse` contains the response to a request (other than a read request) - see Section 34.1.9
  - `sTimerMessage` contains the details of a timer event - this feature is included for future use
  - `sClusterCustomMessage` contains details of a cluster custom command - see Section 34.1.15
  - `sAttributeReportingConfigurationRecord` contains the attribute reporting configuration data from the 'configure reporting' request for an attribute - see Section 34.1.5
  - `sAttributeReportingConfigurationResponse` is reserved for future use
  - `sAttributeDiscoveryResponse` contains the details of an attribute reported in a 'discover attributes' response - see Section 34.1.10
  - `sReportingConfigurationResponse` is reserved for future use
  - `sReportAttributeMirror` contains information on the device from which a ZCL 'report attribute' command has been received
  - `u32TimerPeriodMs` contains the timed period of the millisecond timer which is enabled by the application when the event E_ZCL_CBET_ENABLE_MS_TIMER occurs
  - `sEZBindDetails` is only available if the EZ-mode Commissioning module is enabled (EZ_MODE_COMMISSIONING is TRUE) and contains details of a binding made with a cluster on a remote endpoint - see Section 32.9
  - `sEZGroupDetails` is only available if the EZ-mode Commissioning module is enabled (EZ_MODE_COMMISSIONING is TRUE) and contains details of the addition of a remote endpoint to a group - see Section 32.9
  - `sCommandsReceivedDiscoveryIndividualResponse` contains information about an individual command (that can be received) reported in a Command Discovery response - see Section 34.1.17
  - `sCommandsReceivedDiscoveryResponse` contains information about a Command Discovery response which reports commands that can be recieved - see Section 34.1.18
  - `sCommandsGeneratedDiscoveryIndividualResponse` contains information about an individual command (that can be generated) reported in a Command Discovery response - see Section 34.1.17
  - `sCommandsGeneratedDiscoveryResponse` contains information about a Command Discovery response which reports commands that can be generated - see Section 34.1.18
  - `sAttributeDiscoveryExtenedResponse` contains information from a Discover Attributes Extended response - see Section 34.1.11

Note that sclusterCustomMessage is triggered ony for customized cluster. The inclusion of the structure is described below.



34.1.15 tsZCL_ClusterCustomMessage

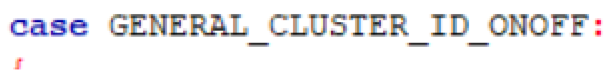
This structure contains a cluster custom message:

```
typedef struct {
     uint16                        u16ClusterId;
     void                          *pvCustomData;
} tsZCL_ClusterCustomMessage;
```

where:

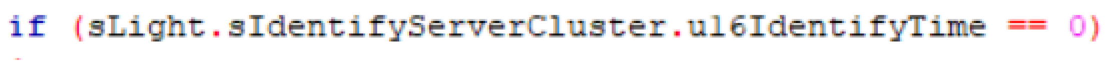
- `u16ClusterId` is the Cluster ID
- `pvCustomData` is a pointer to the start of the data contained in the message

The cluster ID is a 16 bit ID that defines the type of cluster say, in this case our favorable cluster is ON/OFF cluster.



```
case GENERAL_CLUSTER_ID_ONOFF:
```

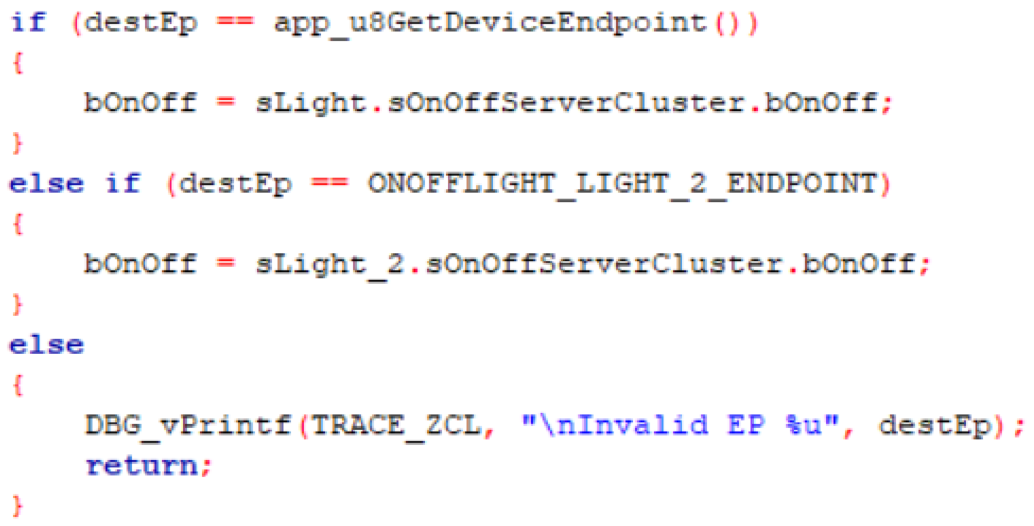
GENERAL_CLUSTER_ID_ON_OFF gives cluster ID or our custom ON_OFF cluster.



```
if (sLight.sIdentifyServerCluster.u16IdentifyTime == 0)
```

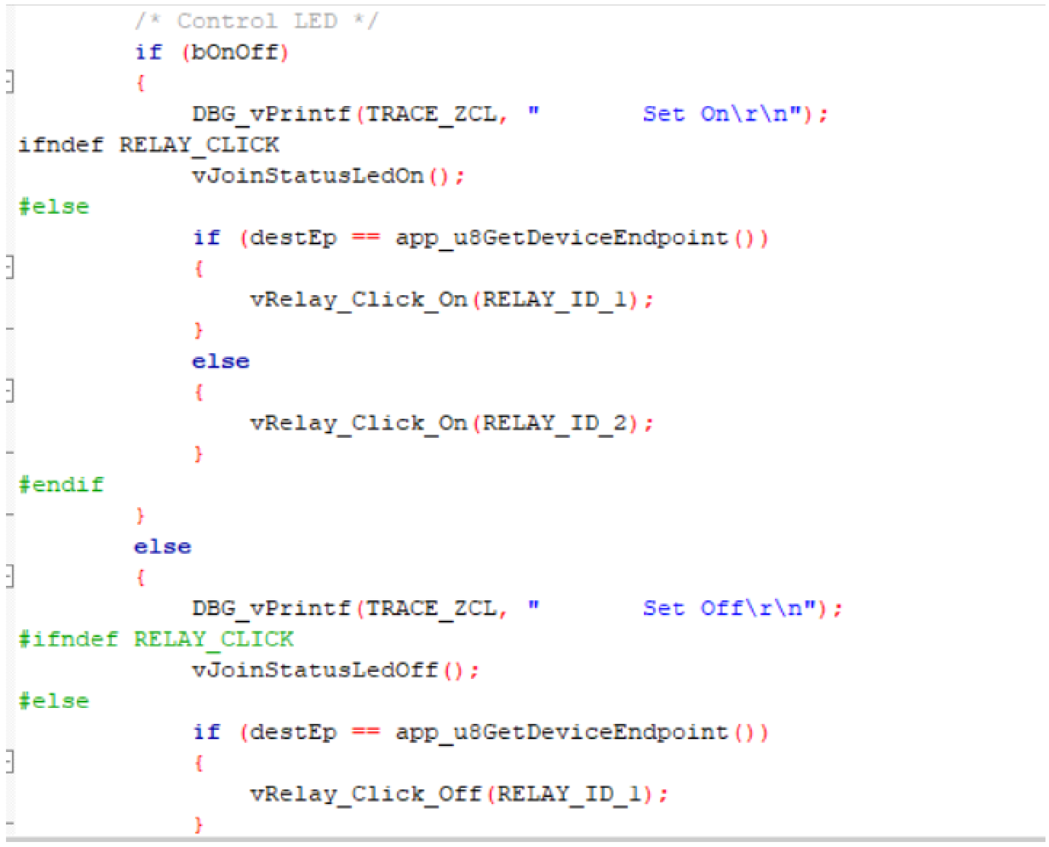
Finally our required relay switching function will be called if the above condition is satisfied. U16IdentifyTime refers to the time required to identify the server and the value of this 16 bit variable shows the remaining time left. The given condition tests if this time left is zero ie..if the system have finished identifying the server cluster.
Now we have to identify how ***APP_ZCL_cbEndpointCallback(tsZCL_CallBackEvent *psEvent)*** function is invoked.

eApp_ZLO_RegisterEndpoint(&APP_ZCL_cbEndpointCallback);
Using this call the previous function is invoked using function pointer.
Explore vhandleOnOffFromZCL(uint8_t destEp) again....



```
if (destEp == app_u8GetDeviceEndpoint())
{
    bOnOff = sLight.sOnOffServerCluster.bOnOff;
}
else if (destEp == ONOFFLIGHT_LIGHT_2_ENDPOINT)
{
    bOnOff = sLight_2.sOnOffServerCluster.bOnOff;
}
else
{
    DBG_vPrintf(TRACE_ZCL, "\nInvalid EP %u", destEp);
    return;
}
```

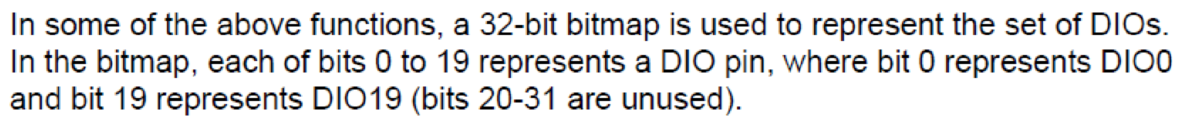
The first if tells that if destination endpoint received upon a ZCL event is relay1 (app_u8GetDeviceEndpoint() returns the address of endpoint 1) , then required state is stored in (bOnOff) .
Now depending upon the value of bOnOff, ie high or low the corresponding endpoint is switched ON or OFF.



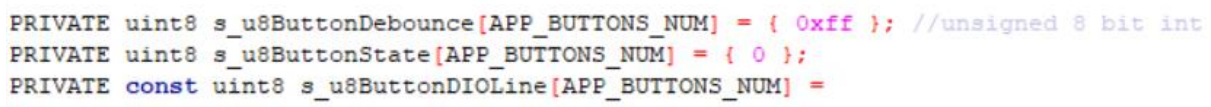
```
        /* Control LED */
        if (bOnOff)
        {
            DBG_vPrintf(TRACE_ZCL, "       Set On\r\n");
ifndef RELAY_CLICK
            vJoinStatusLedOn();
#else
            if (destEp == app_u8GetDeviceEndpoint())
            {
                vRelay_Click_On(RELAY_ID_1);
            }
            else
            {
                vRelay_Click_On(RELAY_ID_2);
            }
#endif
        }
        else
        {
            DBG_vPrintf(TRACE_ZCL, "       Set Off\r\n");
#ifndef RELAY_CLICK
            vJoinStatusLedOff();
#else
            if (destEp == app_u8GetDeviceEndpoint())
            {
                vRelay_Click_Off(RELAY_ID_1);
            }
```

Button Press Handling

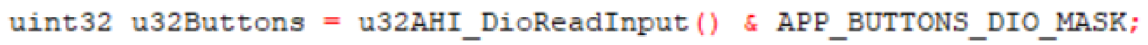
Before learning about button-press events understand the bitmap terminology of DIO pins of JN5179.


In some of the above functions, a 32-bit bitmap is used to represent the set of DIOs. In the bitmap, each of bits 0 to 19 represents a DIO pin, where bit 0 represents DIO0 and bit 19 represents DIO19 (bits 20-31 are unused).

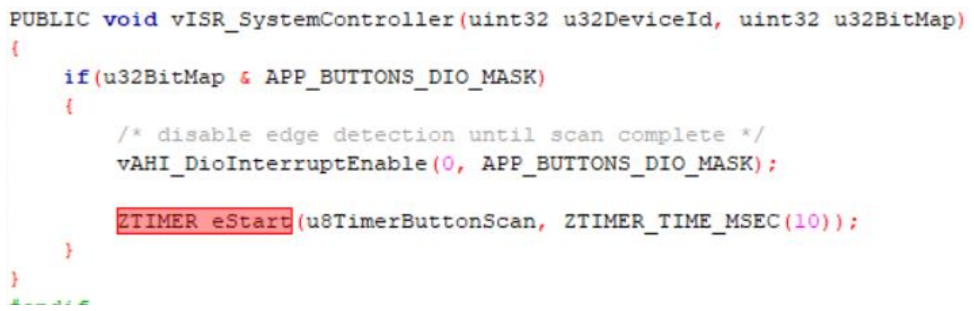
App_buttonhandler.c


```
PRIVATE uint8 s_u8ButtonDebounce[APP_BUTTONS_NUM] = { 0xff }; //unsigned 8 bit int
PRIVATE uint8 s_u8ButtonState[APP_BUTTONS_NUM] = { 0 };
PRIVATE const uint8 s_u8ButtonDIOLine[APP_BUTTONS_NUM] =
```

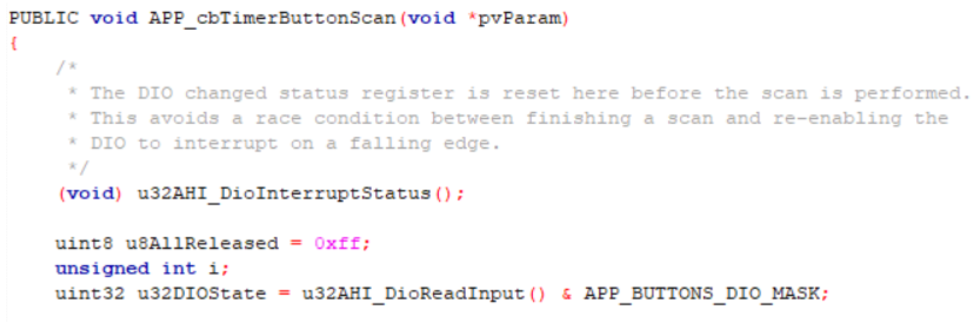
The first ButtonDebounce function is set to ff so that all 8bits are set to 1.
ButtonState function is used to see the state(LOW or HIGH) of the button DIO pin.


```
uint32 u32Buttons = u32AHI_DioReadInput() & APP_BUTTONS_DIO_MASK;
```

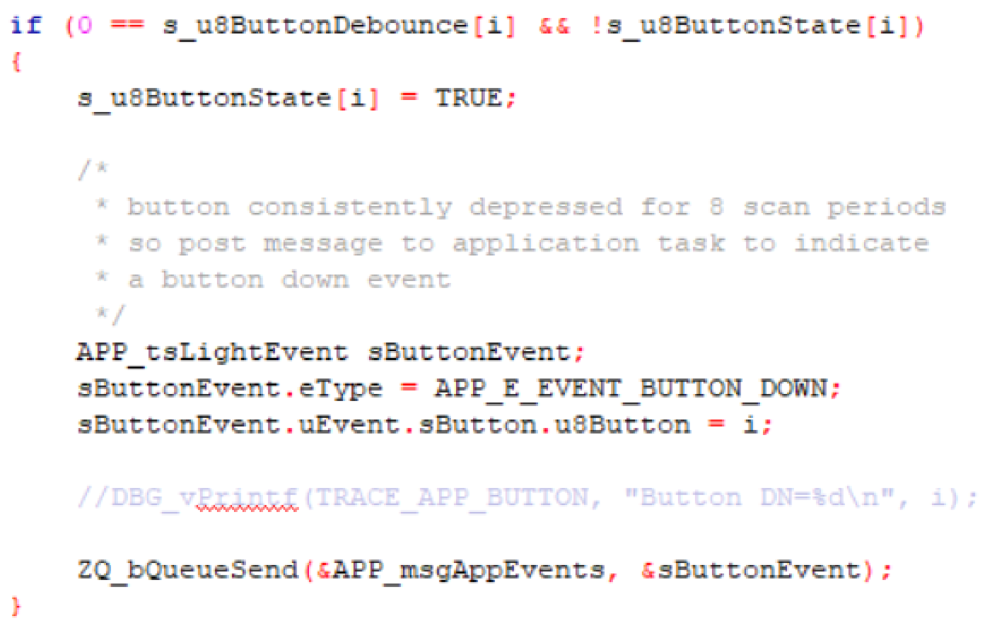
The DioReadInput() will return the state of pin in a 32 bit bit-map form as described above. So if the DIO pin corresponding to the button is active ie if the switch is pressed, u32 buttons will have the bits of the corresponding button/buttons in the Bit-map acive.


```
PUBLIC void vISR_SystemController(uint32 u32DeviceId, uint32 u32BitMap)
{
    if(u32BitMap & APP_BUTTONS_DIO_MASK)
    {
        /* disable edge detection until scan complete */
        vAHI_DioInterruptEnable(0, APP_BUTTONS_DIO_MASK);

        ZTIMER_eStart(u8TimerButtonScan, ZTIMER_TIME_MSEC(10));
    }
}
```

This is the interrupt service routine once the interrupt is triggered. If the bit-map corresponding to triggered pin is same as the bitmap of button pin, The interrupt corresponding to that pin will be temporarily deactivated and a ZTIMER_eStart function is called which is probably a scan routine to scan the state of pin..which we will explain later.

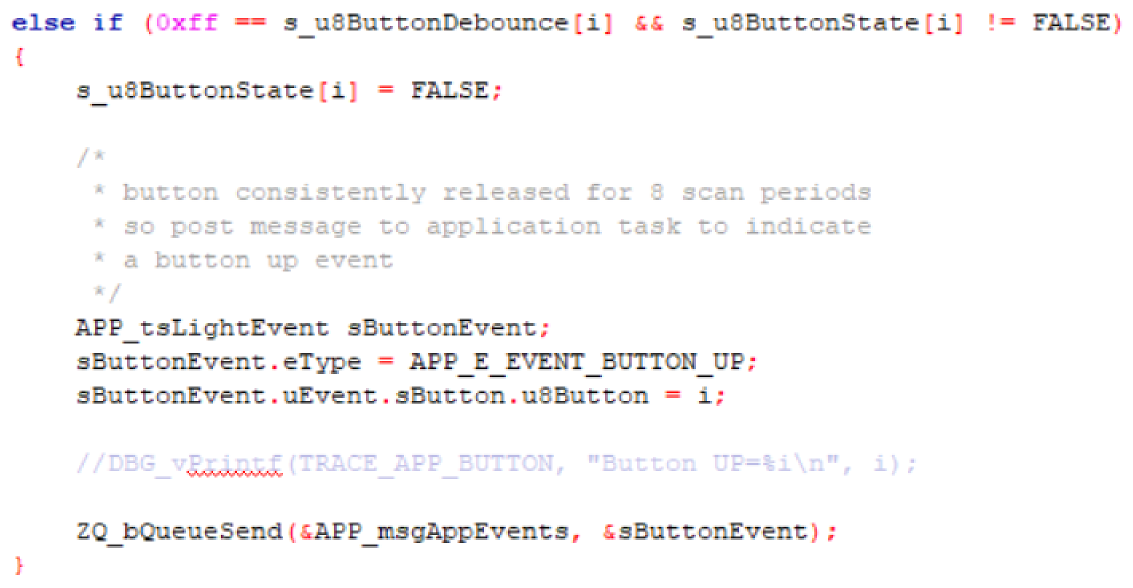

```
PUBLIC void APP_cbTimerButtonScan(void *pvParam)
{
    /*
     * The DIO changed status register is reset here before the scan is performed.
     * This avoids a race condition between finishing a scan and re-enabling the
     * DIO to interrupt on a falling edge.
     */
    (void) u32AHI_DioInterruptStatus();

    uint8 u8AllReleased = 0xff;
    unsigned int i;
    uint32 u32DIOState = u32AHI_DioReadInput() & APP_BUTTONS_DIO_MASK;
```

This is probably the button scan routine which triggers the right button_event corresponding to the type of activity.



```
if (0 == s_u8ButtonDebounce[i] && !s_u8ButtonState[i])
{
    s_u8ButtonState[i] = TRUE;

    /*
     * button consistently depressed for 8 scan periods
     * so post message to application task to indicate
     * a button down event
     */
    APP_tsLightEvent sButtonEvent;
    sButtonEvent.eType = APP_E_EVENT_BUTTON_DOWN;
    sButtonEvent.uEvent.sButton.u8Button = i;

    //DBG_vPrintf(TRACE_APP_BUTTON, "Button DN=%d\n", i);

    ZQ_bQueueSend(&APP_msgAppEvents, &sButtonEvent);
}
```

Now this is the next significant if statement which signifies a button press event. The condition within the if statement basically ensures button debounce and checks if the previous button state was OFF state the verify that it is a button-press event.
Once the event is verified, a structure named sButtonEvent is created which contains the profile of the button pressed and the type of event, here, a button press event. Then it is sent to a zigbee queue so that the message can be received by the function checking the event.



```
else if (0xff == s_u8ButtonDebounce[i] && s_u8ButtonState[i] != FALSE)
{
    s_u8ButtonState[i] = FALSE;

    /*
     * button consistently released for 8 scan periods
     * so post message to application task to indicate
     * a button up event
     */
    APP_tsLightEvent sButtonEvent;
    sButtonEvent.eType = APP_E_EVENT_BUTTON_UP;
    sButtonEvent.uEvent.sButton.u8Button = i;

    //DBG_vPrintf(TRACE_APP_BUTTON, "Button UP=%i\n", i);

    ZQ_bQueueSend(&APP_msgAppEvents, &sButtonEvent);
}
```

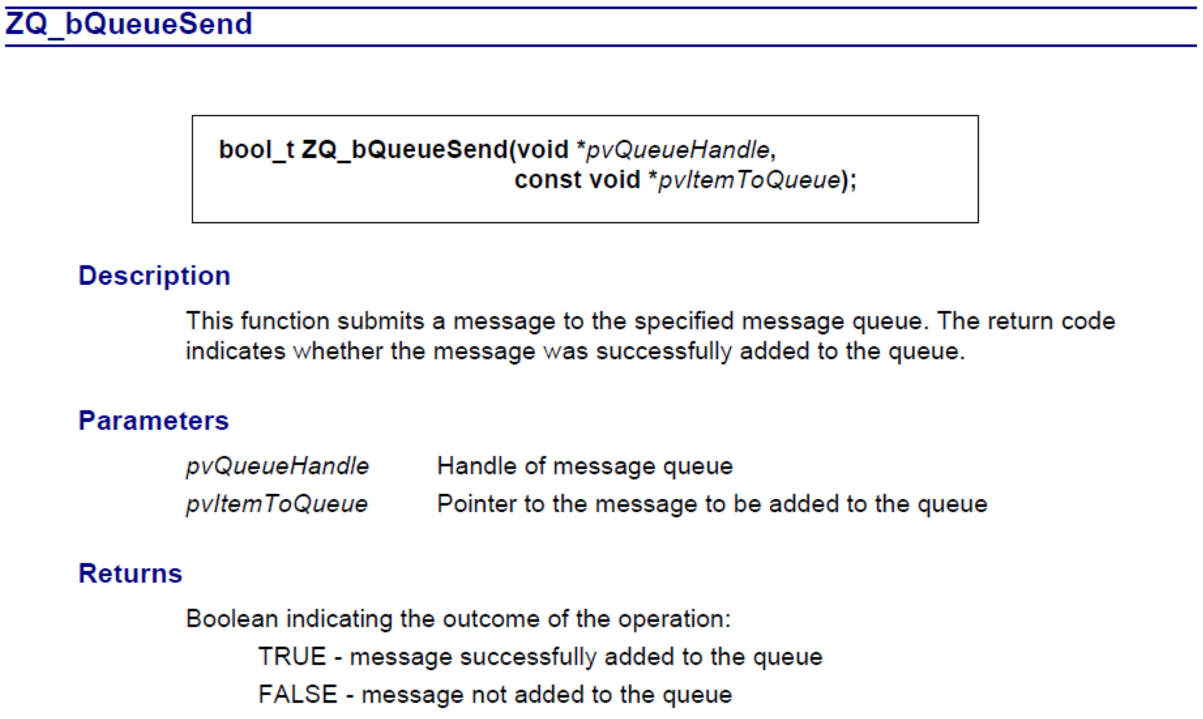
This is the counter else of the previous if statement for a button release event it does the same process as the if statement except the event type is set to button release event.

PUBLIC void APP_taskLight(void)
(app_zlo_lightnode.c)

This function is responsible for receiving an event information related to any button press. For our work, it checks the eventqueue for a button press and in case of a button press the button pres value is incremented and sent to the connected master node for display in application.

Message Queue

Before proceeding further on this function we need to thoroughly understand about dealing with message queue and software timers.



ZQ_bQueueSend

```
bool_t ZQ_bQueueSend(void *pvQueueHandle,
                     const void *pvItemToQueue);
```

Description

This function submits a message to the specified message queue. The return code indicates whether the message was successfully added to the queue.

Parameters

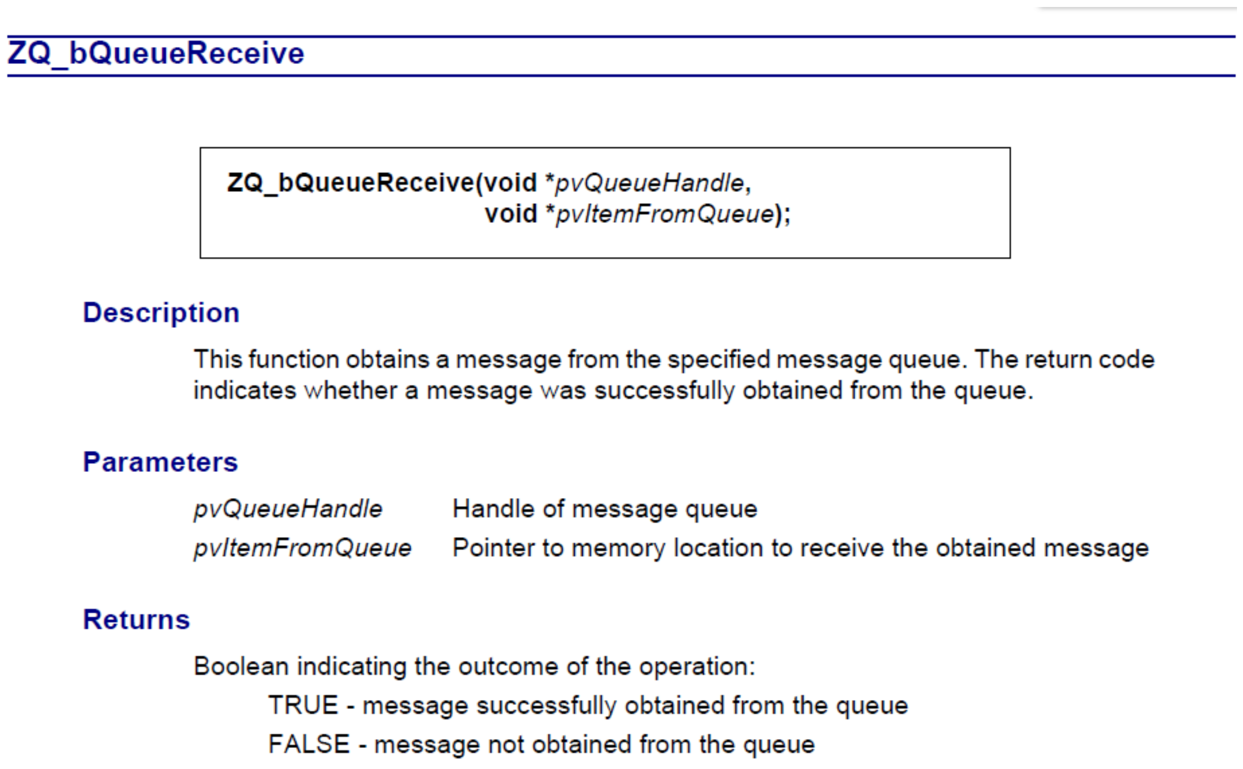
pvQueueHandle	Handle of message queue
pvItemToQueue	Pointer to the message to be added to the queue

Returns

Boolean indicating the outcome of the operation:

TRUE - message successfully added to the queue

FALSE - message not added to the queue



ZQ_bQueueReceive

```
ZQ_bQueueReceive(void *pvQueueHandle,
                 void *pvItemFromQueue);
```

Description

This function obtains a message from the specified message queue. The return code indicates whether a message was successfully obtained from the queue.

Parameters

pvQueueHandle	Handle of message queue
pvItemFromQueue	Pointer to memory location to receive the obtained message

Returns

Boolean indicating the outcome of the operation:

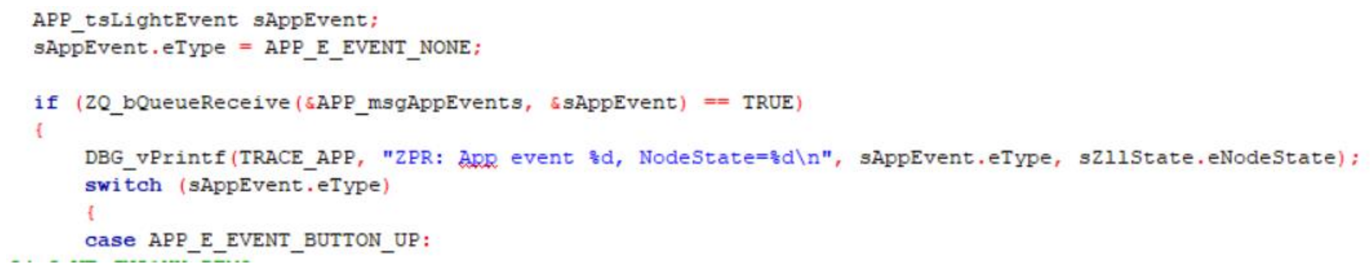
TRUE - message successfully obtained from the queue

FALSE - message not obtained from the queue

ZQ_bQueueIsEmpty()

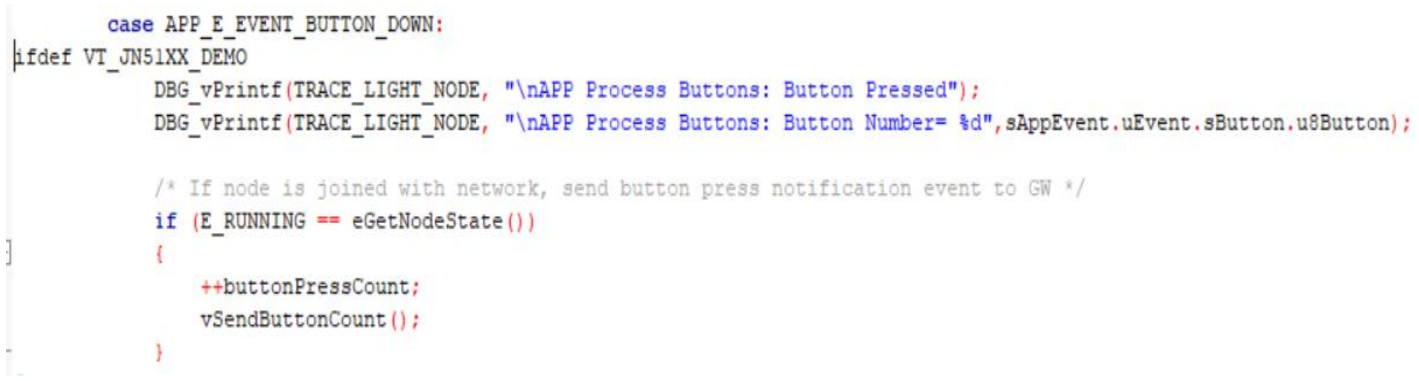
The above function checks if the queue is empty.

Now back to function



```
APP_tsLightEvent sAppEvent;
sAppEvent.eType = APP_E_EVENT_NONE;

if (ZQ_bQueueReceive(&APP_msgAppEvents, &sAppEvent) == TRUE)
{
    DBG_vPrintf(TRACE_APP, "ZPR: App event %d, NodeState=%d\n", sAppEvent.eType, sZllState.eNodeState);
    switch (sAppEvent.eType)
    {
    case APP_E_EVENT_BUTTON_UP:
```

This checks if there is a message in the message queue and checks the type of event.



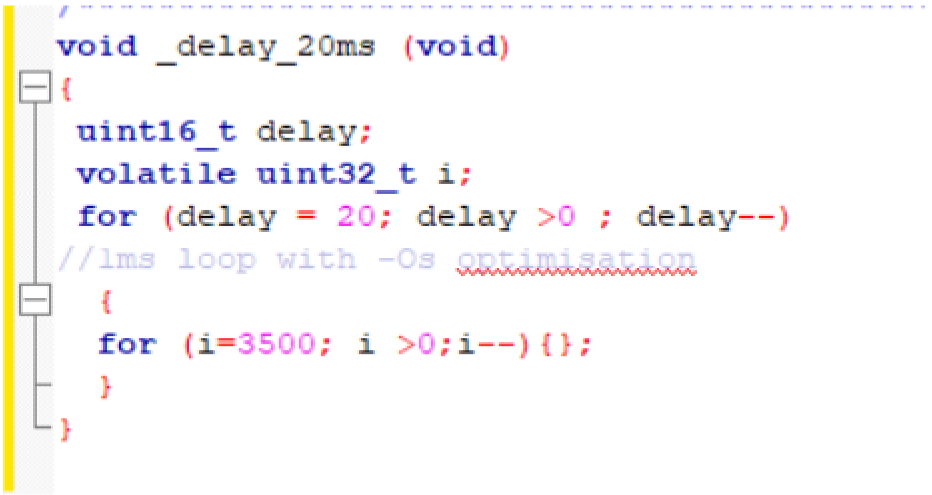
```
        case APP_E_EVENT_BUTTON_DOWN:
ifdef VT_JN51XX_DEMO
            DBG_vPrintf(TRACE_LIGHT_NODE, "\nAPP Process Buttons: Button Pressed");
            DBG_vPrintf(TRACE_LIGHT_NODE, "\nAPP Process Buttons: Button Number= %d",sAppEvent.uEvent.sButton.u8Button);

            /* If node is joined with network, send button press notification event to GW */
            if (E_RUNNING == eGetNodeState())
            {
                ++buttonPressCount;
                vSendButtonCount();
            }
```

In case of a button down event, the button press count is incremented and sent to the aoo using vSendButtonCount().

Modified Functions

In the above function any button up event is followed by a button down event.Inorder to transmit a data encrypted in time you can search for a button down event immediately as you detect one clean the immediate next event as the possibility is that the immediate previous even is a button up event and this might ruin your function. After this wait until for a button up event and once you detect it stop your timer and send the timer count.

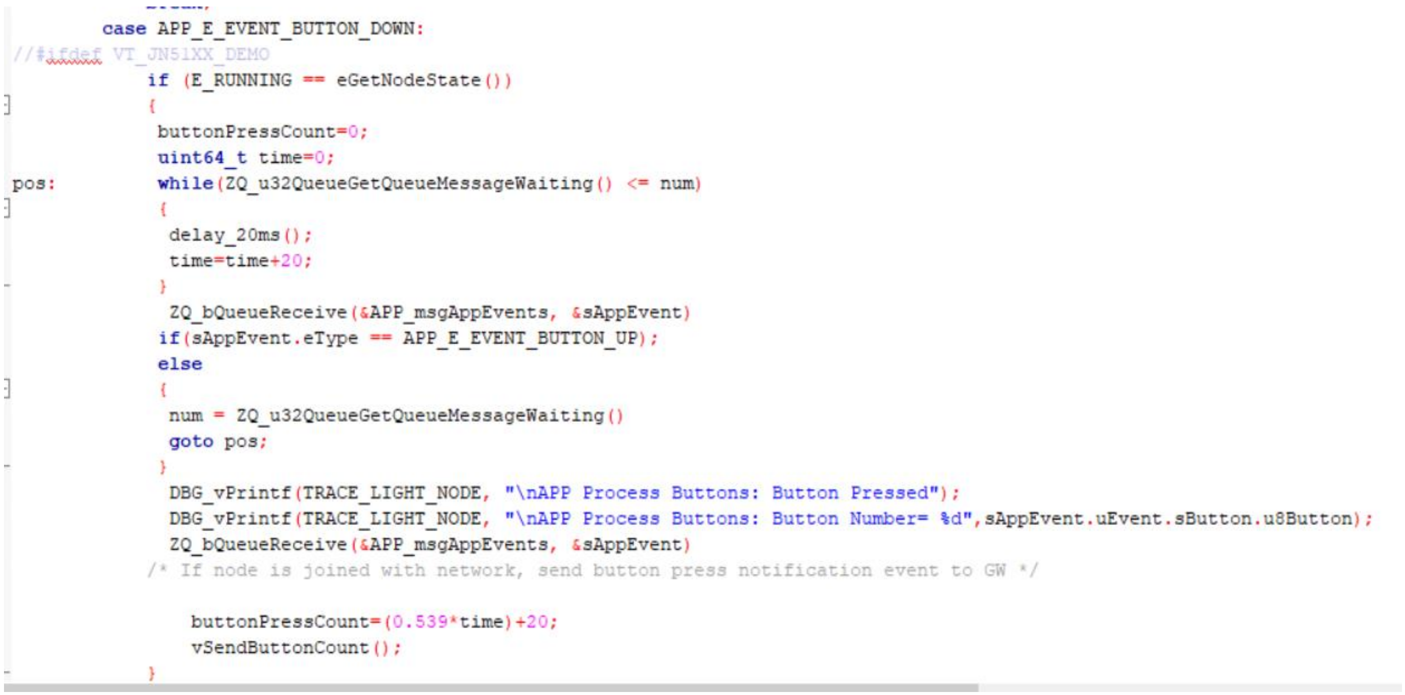
Timing of Button-Press using software timing



```
void _delay_20ms (void)
{
 uint16_t delay;
 volatile uint32_t i;
 for (delay = 20; delay >0 ; delay--)
//1ms loop with -Os optimisation
  {
  for (i=3500; i >0;i--){};
  }
}
```

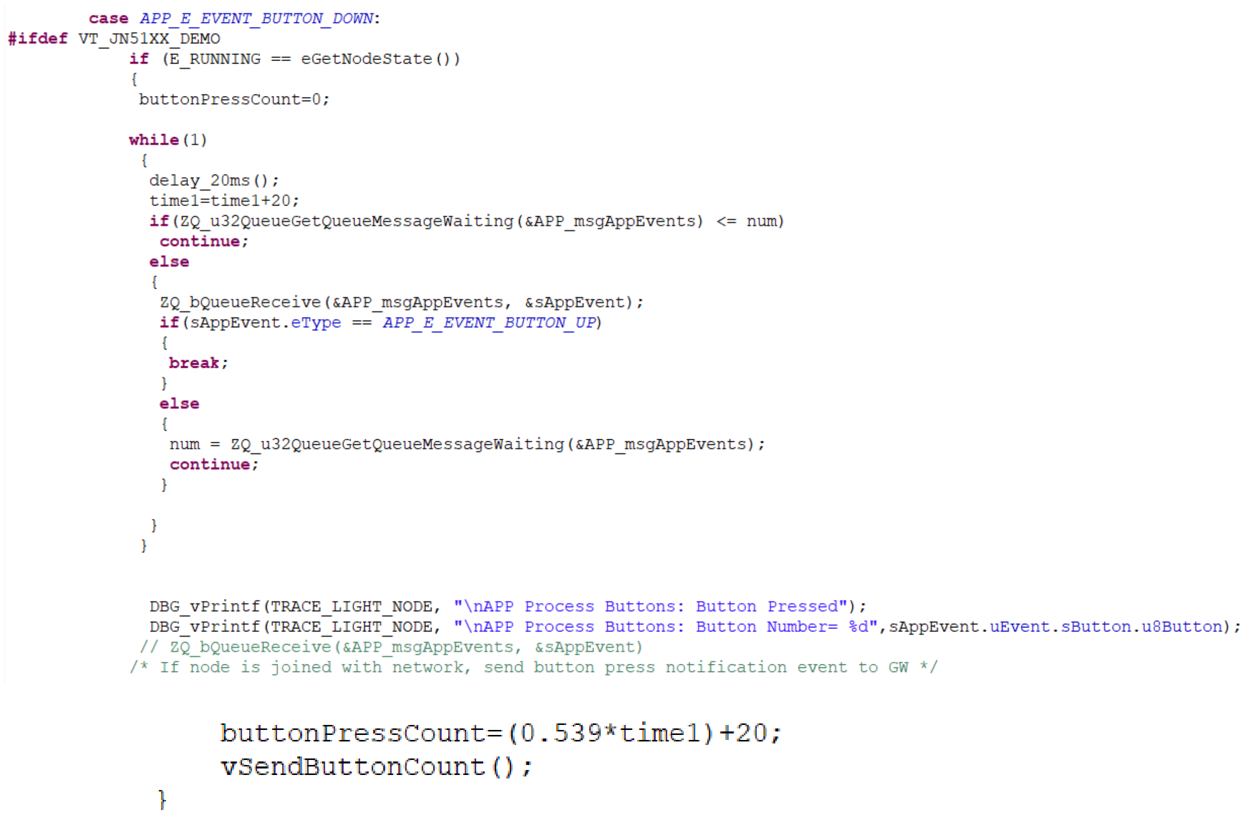
ZQ_u32QueueGetQueueMessageWaiting() can be used to check if a new message is added to the queue. If it is added our timer has to be teminated

The modified button-press response



```
        case APP_E_EVENT_BUTTON_DOWN:
//#ifdef VT_JN51XX_DEMO
            if (E_RUNNING == eGetNodeState())
            {
             buttonPressCount=0;
             uint64_t time=0;
pos:         while(ZQ_u32QueueGetQueueMessageWaiting() <= num)
             {
              delay_20ms();
              time=time+20;
             }
              ZQ_bQueueReceive(&APP_msgAppEvents, &sAppEvent)
             if(sAppEvent.eType == APP_E_EVENT_BUTTON_UP);
             else
             {
              num = ZQ_u32QueueGetQueueMessageWaiting()
              goto pos;
             }
              DBG_vPrintf(TRACE_LIGHT_NODE, "\nAPP Process Buttons: Button Pressed");
              DBG_vPrintf(TRACE_LIGHT_NODE, "\nAPP Process Buttons: Button Number= %d",sAppEvent.uEvent.sButton.u8Button);
              ZQ_bQueueReceive(&APP_msgAppEvents, &sAppEvent)
            /* If node is joined with network, send button press notification event to GW */

                buttonPressCount=(0.539*time)+20;
                vSendButtonCount();
            }
```

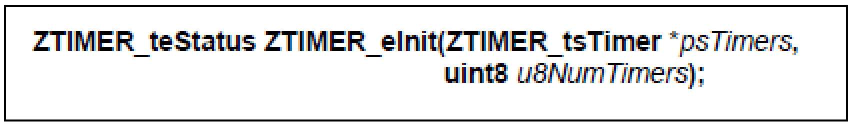
As you can see once a button press event is recognized, immediately the button press value is initialized to zero and a variable named time is declared to measure time intervals of 20 milliseconds.
If a new message is added to message queue via an interrupt, the while loop breaks.
The next few statements checks if the added even is a button up event else the timer is continued.
Now the final time count is sent via an encryption formula as shown in the code.
LCP Expresso Debugged Version



```
        case APP_E_EVENT_BUTTON_DOWN:
#ifdef VT_JN51XX_DEMO
            if (E_RUNNING == eGetNodeState())
            {
             buttonPressCount=0;

            while(1)
             {
              delay_20ms();
              time1=time1+20;
              if(ZQ_u32QueueGetQueueMessageWaiting(&APP_msgAppEvents) <= num)
               continue;
              else
              {
               ZQ_bQueueReceive(&APP_msgAppEvents, &sAppEvent);
               if(sAppEvent.eType == APP_E_EVENT_BUTTON_UP)
               {
                break;
               }
               else
               {
                num = ZQ_u32QueueGetQueueMessageWaiting(&APP_msgAppEvents);
                continue;
               }

              }
             }


              DBG_vPrintf(TRACE_LIGHT_NODE, "\nAPP Process Buttons: Button Pressed");
              DBG_vPrintf(TRACE_LIGHT_NODE, "\nAPP Process Buttons: Button Number= %d",sAppEvent.uEvent.sButton.u8Button);
             // ZQ_bQueueReceive(&APP_msgAppEvents, &sAppEvent)
            /* If node is joined with network, send button press notification event to GW */

                    buttonPressCount=(0.539*time1)+20;
                    vSendButtonCount();
             }
```

Unfortunately the above code doesn't work on hardware..So to implement timing mechanism the only way is software timers.

Software Timers



```
ZTIMER_teStatus ZTIMER_eInit(ZTIMER_tsTimer *psTimers,
                             uint8 u8NumTimers);
```

This function in ztimers.h is used to initialize a series of software timers. The first parameter is the pointer to an array of structures, the structure is defined by,

```
typedef struct
{
    ZTIMER_teState      eState;
    uint32              u32Time;
    void               *pvParameters;
    ZTIMER_tpfCallback  pfCallback;
} ZTIMER_tsTimer;
```

The first parameter is the state of the timer (not significant for our program)
Second one, u32Time is the remaining time for which the timer has to run is the significant part of our program
pfCallback is the user defined call back function, that has to be called once the timer expires.
u8Numtimers is the number of software timers defined.

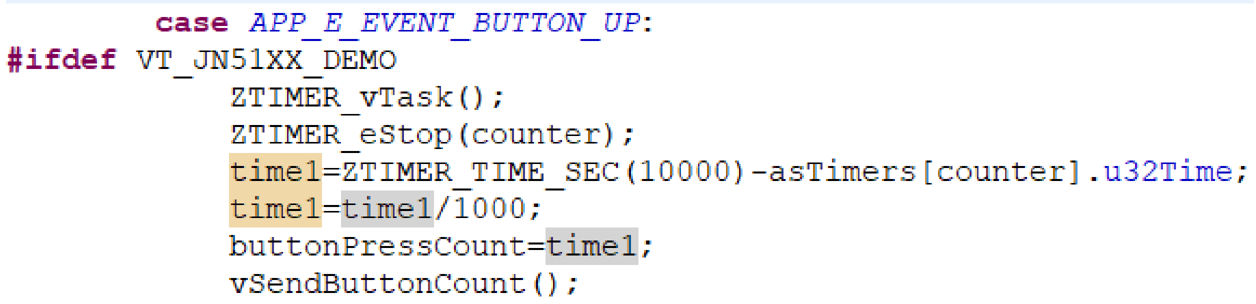
```
ZTIMER_teStatus ZTIMER_eOpen(
                             uint8 *pu8TimerIndex,
                             ZTIMER_tpfCallback pfCallback,
                             void *pvParams,
                             uint8 u8Flags);
```

Used to open the specified timer
The first parameter is a pointer to the location containing the index of timer.
pfCallback is a pointer to the function which has to be called once the timer expires. In our program you can return NULL.
Pvparams is the timer parameters, again has to be set to NULL.
The u8flag is a flag which shows the behavior of the controller when timer is active ie..whether it should go to sleep or not. You should give ZTIMER_FLAG_PREVENT_SLEEP. As the parameter.

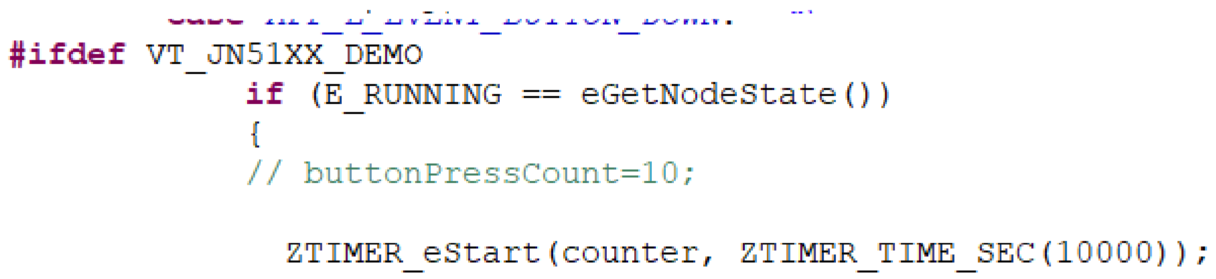
```
ZTIMER_teStatus ZTIMER_eStart(uint8 u8TimerIndex,
                              uint32 u32Time);
```

The first parameter is the index number of the timer which is initialized. Second one is the time in milliseconds for which the timer should run.

Successful Code Segments



```
        case APP_E_EVENT_BUTTON_UP:
#ifdef VT_JN51XX_DEMO
            ZTIMER_vTask();
            ZTIMER_eStop(counter);
            time1=ZTIMER_TIME_SEC(10000)-asTimers[counter].u32Time;
            time1=time1/1000;
            buttonPressCount=time1;
            vSendButtonCount();
```



```
#ifdef VT_JN51XX_DEMO
            if (E_RUNNING == eGetNodeState())
            {
            // buttonPressCount=10;

               ZTIMER_eStart(counter, ZTIMER_TIME_SEC(10000));
```