Z-Stack
Sample Applications

Document Number: SWRA201

Texas Instruments, Inc.
San Diego, California USA

Version	Description	Date
1.0	Initial release.	12/13/2006
1.1	Updated key press descriptions for SampleLight application.	05/21/2007
1.2	Updated to new OSAL task and ZDO design	06/23/2008
1.3	Updated for Z-Stack 2.2.0 release	4/1/2009

TABLE OF CONTENTS

i. Acronyms

API	Application Programming Interface
APL	Application Layer
APS	ZigBee Application Support Sublayer
BSP	Board Support Package – taken together, HAL & OSAL comprise a rudimentary operating system commonly referred to as a BSP.
EP	Endpoint
HAL	Hardware (H/W) Abstraction Layer
HA	Home Automation (A Stack Profile, Profile Id 0x0104)
MT	Monitor Test
NWK	ZigBee Network Layer
OSAL	Operating System (OS) Abstraction Layer
OTA	Over-The-Air
PC	Personal Computer
SAP	Service Access Point
SPI	Serial Port Interface
ZCL	ZigBee Cluster Library
ZDO	ZigBee Device Objects

1. Z-Stack Sample Applications

1.1 Introduction

This document covers the 4 simple Z-Stack™ Sample Applications: Generic, Serial, Transmit, & HomeAutomation. Each one of the Z-Stack Sample Applications is a simple head-start to using the TI distribution of the ZigBee Stack in order to implement a specific Application Object.

Each sample application uses the minimal subset of ZDO Public Interfaces that it would take to make a Device reasonably viable in a ZigBee network. In addition, all sample applications utilize the essential OSAL API functionality: inter and intra-task communication by sending and receiving messages, setting and receiving task events, setting and receiving timer callbacks, using dynamic memory, as well as others. And every sample application makes use of the HAL API functionality by controlling LED's. Thus, any sample application serves as a fertile example from which to copy-and-paste, or as the base code of the user's application to which to add additional Application Objects.

By definition, an Application Object can only support a single Profile; and each of the Z-Stack Sample Applications implements an unofficial Private Profile. The Private Profile Id's used by the sample applications have been arbitrarily chosen and must not be used outside of the development laboratory.[1] Although a Private Profile may use a unique network configuration, the sample applications have been implemented and tested with the network configuration specified by the Stack Profile known as HomeControlLighting (0x0100.)

Any Application Object must sit overtop of a unique Endpoint; and any Endpoint is defined by a Simple Descriptor. The numbers used for the Endpoints in the Simple Descriptors in the sample applications have been chosen arbitrarily.

Each sample application instantiates only one Application Object, and therefore, only supports the one corresponding Profile. But keep in mind that two or more Application Objects may be instantiated in the same Device. When instantiating more than one Application Object in the same Device, each Application Object must implement a unique Profile Id and sit overtop of a unique Endpoint number. The sample applications meet the unique Id's and Endpoint numbers requirement and could be combined into one Device with minor modifications.[2]

1.2 OSAL Tasks

1.2.1 Initialization

OSAL is designed and distributed as source so that the entire OSAL functionality may be modified by the Z-Stack user. The goal of the design is that it should not be necessary to modify OSAL in order to use the Z-Stack as distributed. The one exception is that the OSAL Task initialization function, osalInitTasks(), must be implemented by the user. The sample applications have implemented said function in a dedicated file named something like this: OSAL_"Application Name".c (e.g. OSAL_GenericApp.c). The BSP will invoke osalInitTasks() as part of the board power-up and Z-Stack initialization process.

1.2.2 Organization

As described in the Z-Stack OSAL API (SWRA194) document, the OSAL implements a cooperative, round-robin task servicing loop. Each major sub-system of the Z-Stack runs as an OSAL Task. The user must create at least one OSAL Task in which their application will run. This is accomplished by adding their task to the task array [tasksArr defined in OSAL_"Application Name".c] and calling their application's task initialization function in osalInitTask(). The sample applications clearly show how the user adds a task to the OSAL system.

1.2.3 Task Priority

The tasks are executed in their order of placement in the task array [tasksArr in OSAL_"Application Name".c]. The first task in the array has the highest priority.

1.2.4 System Services

OSAL and HAL system services are exclusive - only one OSAL Task may register for keyboard (switch press) notification and serial port activity notification. The same task does not have to register for both, and none of the Z-Stack Tasks register for either – they are both left to the user's application.

1.2.5 Application Design

The user may create one task for each Application Object instantiated or service all of the Application Objects with just one task. The following are some of the considerations when making the aforementioned design choice.

1.2.5.1 One OSAL Task to many Application Objects

These are some of the pros & cons of the one-to-many design:

- Pro: The action taken when receiving an exclusive task event (switch press or serial port) is simplified.
- Pro: The heap space required for many OSAL Task structures is saved.
- Con: The action taken when receiving an incoming AF message or an AF data confirmation is complicated – the burden for de-multiplexing the intended recipient Application Object is on the single user task.
- Con: The process of service discovery by Match Descriptor Request (i.e. auto-match) is more complicated – a static local flag must be maintained in order to act properly on the `ZDO_NEW_DSTADDR` message.

1.2.5.2 One OSAL Task to one Application Object

These pros & cons of the one-to-one design are the inverse of those above for the one-to-many design:

- Pro: An incoming AF message or an AF data confirmation has already been de-multiplexed by the lower layers of the stack, so the receiving Application Object is the intended recipient.
- Con: The heap space required for many OSAL Task structures is incurred.
- Con: The action taken when receiving an exclusive task event may be more complicated if two or more Application Objects use the same exclusive resource.

1.2.6 Mandatory Methods

Any OSAL Task must implement two methods: one to perform task initialization and the other to handle task events.

1.2.6.1 Task Initialization

The callback function to perform task initialization is named like this in the sample applications: "Application Name"_Init (e.g. `GenericApp_Init`). The task initialization function should accomplish the following:

- Initialization of variables local to or specific for the corresponding Application Object(s). Any long-lived heap memory allocation should be made in order to facilitate more efficient heap memory management by the OSAL.
- Instantiation of the corresponding Application Object(s) by registering with the AF layer (e.g. `afRegister()`).
- Registration with the applicable OSAL or HAL system services (e.g. `RegisterForKeys()`).

1.2.6.2 Task Event Handler

The callback function to handle task events is named like this in the sample applications: "Application Name"_ProcessEvent (e.g. `GenericApp_ProcessEvent()`). Any OSAL Task can define up to 15 events in addition to the mandatory event.

1.2.7 Mandatory Events

One task event, SYS_EVENT_MSG (0x8000), is reserved and required by the OSAL Task design.

1.2.7.1 SYS_EVENT_MSG (0x8000)

The global system messages sent via the SYS_EVENT_MSG are specified in ZComDef.h. The task event handler should process the following minimal subset of these global system messages. The recommended processing of the following messages should be learned directly from the sample application code or from the study of the program flow in the sample application.

1.2.7.1.1 AF_DATA_CONFIRM_CMD

This is an indication of the OTA result for each data request that is successfully initiated by invoking AF_DataRequest(). ZSuccess confirms that the data request was successfully transmitted OTA. If the data request was made with the AF_ACK_REQUEST flag set, then the ZSuccess confirms that the message was successfully received at the final destination. Otherwise, the ZSuccess only confirms that the message was successfully transmitted to the next hop.[3]

1.2.7.1.2 AF_INCOMING_MSG_CMD

This is an indication of an incoming AF message.

1.2.7.1.3 KEY_CHANGE

This is an indication of a key press action.

1.2.7.1.4 ZDO_STATE_CHANGE

This is an indication that the network state has changed.

1.2.7.1.5 ZDO_CB_MSG

This is message is sent to the sample application for every registered ZDO response message [ZDO_RegisterForZDOMsg()].

1.3 Network Formation

A sample application compiled as a Coordinator will form a network on one of the channels specified by the DEFAULT_CHANLIST. The Coordinator will establish a random Pan ID based on its own IEEE address or on ZDAPP_CONFIG_PAN_ID if it is not defined as 0xFFFF. A sample application compiled as a Router or End Device will try to join a network on one of the channels specified by DEFAULT_CHANLIST. If ZDAPP_CONFIG_PAN_ID is not defined as 0xFFFF, the Router will be constrained to join only the Pan ID thusly defined. Note the unexpected result achieved because of the difference in behavior between a Coordinator and a Router or End Device when ZDAPP_CONFIG_PAN_ID is not defined as 0xFFFF. If ZDAPP_CONFIG_PAN_ID is defined as a valid value less than or equal to 0xFFFE, then the Coordinator will only attempt to establish a network with the specified Pan Id. Thus, if the Coordinator is constrained to one channel, and the specified Pan Id has already been established on that channel, the newly starting Coordinator will make successive changes until it achieves a unique Pan Id. A Router or End Device newly joining will have no way of knowing the value of the "de-conflicted" Pan Id established, and will therefore join only the Pan Id specified. A similarly challenging scenario arises when the permitted channel mask allows more than one channel and the Coordinator cannot use the first channel because of a Pan Id conflict – a Router or End Device will join the specified Pan Id on the first channel scanned, if allowed.

1.3.1 Auto Start

A device will automatically start trying to form or join a network as part of the BSP power-up sequence. If the device should wait on a timer or other external event before joining, then HOLD_AUTO_START must be defined. In order to manually start the join process at a later time, invoke ZDOInitDevice().

1.3.2 Network Restore

Devices that have successfully joined a network can "restore the network" (instead of reforming by OTA messages) even after losing power or battery. This automatic restoration can be enabled by defining `NV_RESTORE` and/or `NV_INIT`.

1.3.3 Join Notification

The device is notified of the status of the network formation or join (or any change in network state) with the `ZDO_STATE_CHANGE` message mentioned above.

1.4 Device Binding & Discovery

The development board has 4 switches, or keys, known as SW1-SW4. The sample applications implement device binding by SW2 and device discovery by SW4. In order to make two matching sample applications talk OTA, there must be a SW2 press on both of them within a short amount of time as well as a Coordinator device present in the network; or there must be a SW4 press on at least one of the devices. On the development kit boards, success of the bind or discovery will be indicated by a new LED turning on. That same LED will flash when the maximum binding time has passed without a success.

1.4.1 Bind Request (Hand Binding)

In response to a SW2 press, the sample applications utilize the ZDO API `ZDApp_SendEndDeviceBindReq()` in order to participate in the device hand binding process. A reflector device (usually the coordinator) will receive the bind request. Another device in the network with one or more input clusters that match one or more of the first device's output clusters (or vice versa: output clusters matching input clusters) must also make a `ZDApp_SendEndDeviceBindReq()`. Such a request must be made within `APS_DEFAULT_MAXBINDING_TIME` milliseconds so that a successful match can be made (i.e. the network reflector will not hold onto the first `ZDApp_SendEndDeviceBindReq()` forever, awaiting a successful match.) These are some of the pros & cons of using binding:

- Pro: The binding information resides on the network reflector device, saving RAM space on the target devices.
- Pro: The network reflector must be `RxOnWhenIdle` (always listening to the network). So if one of the bound devices broadcasts that its network address has changed, then the network reflector will update the corresponding binding table entry. Thus, even if the other bound device was sleeping, its subsequent messages to the changed device will still be correctly addressed by the network reflector.
- Con: A device that is bound to more than one other device cannot send a message to one or a subset of its matching devices – the reflector will generate a unicast message to all of the bound devices.
- Con: A sending device cannot receive notification of delivery to the destination device (i.e. the expected result when using the `AF_ACK_REQUEST` flag.)
- Con: All messages must go thru the network reflector, reducing network bandwidth.
- Pro/Con: A sending device bound to six devices will send one message to the network reflector which will result in the network reflector sending six unicast messages. Consider a network divided into two equal geographic regions, A and B, with the network reflector on the centerline. If the sending device is deep into region A and the receiving devices are deep into region B, then for every message sent, network traffic in region A will be six times less than it would be if the sending device has generated six instead of one trans-network messages. If all devices are nearby each other, deep in one region, then network traffic in that region is very much more than if the sending device could just send single hop messages to the intended recipients.

1.4.2 Match Descriptor Request (Auto Find)

As implemented in the sample applications, this feature is a way to find a device that accepts (as input) what this device outputs. The sample application will do the following steps (in order):

- Call ZDO_RegisterForZDOMsg() to register with ZDO that it wants to process incoming ZDO Match Descriptor Response messages.
- Call ZDP_MatchDescReq() with a list of output clusters (msgs) that it supports. This will send a broadcast message to all devices looking for a match. Matching devices will respond with unicast messages.
- Process incoming ZDO Match Descriptor Response messages. ZDO will send all over-the-air response messages, the application registered for these with ZDO_RegisterForZDOMsg(), as an OSAL message (ZDO_CB_MSG). The application will process these response messages and save the message's source address as the applications default destination address (dstAddr) for any outgoing message.

In the sample application, pressing SW4 will call ZDP_MatchDescReq(). The pros & cons of using Auto Find are:

- Con: The destination information resides on the finding device. This local information requires not only RAM space on the target devices, but possibly non-volatile memory and backup and restore logic in order to not lose discovery information after a power cycle. Also, the way it's implemented in the sample applications, only one destination address is kept (the last one). Option: the user can easily change the application to keep multiple destination addresses.
- Con: The finding device must be RxOnWhenIdle (always listening to the network) if it is ever possible for the devices discovered to change their network address.
- Pro: Responses received during finding can be filtered and not kept. If many matching responses are kept, a device can always choose any one or subset of the matches when sending OTA messages.
- Pro: A sending device will receive notification of delivery to the destination device (i.e. the expected result when using the AF_ACK_REQUEST flag.)
- Pro: Messages are routed by best path, conserving network bandwidth.
- Pro/Con: A finding device that has six matching responses will have to send six unicast messages. Consider a network divided into two equal geographic regions, A and B, with the network reflector on the centerline. If the sending device is deep into region A and the receiving devices are deep into region B, then all six messages will impact network throughput in both regions. If all devices are nearby each other, deep in one region, then network traffic in the other region is completely unaffected.

1.5 Common Application Framework

This section describes the initialization and main task processing concepts that all sample applications share. This section should be read before moving on to the sample applications. For code examples in this section, we will use GenericApp.

1.6 Program Flow

1.6.1 Initialization

During system power-up and initialization[4] the task's task initialization function (see 1.2.6.1 Task Initialization) will be invoked.

```
GenericApp_TaskID = task_id;
```


Notice that OSAL assigns the task its Task Id via the function parameter. This is the Task Id that the sample application must use to set a timer for itself, to set an event for itself, or to send an OSAL message to itself. Any of the aforementioned operations might be done in order to divide a large chunk processing up into smaller chunks that are executed on successive "time slots" from OSAL instead of taking too much time on any single time slot. When a task divides a large chunk of work into smaller chunks that are executed one per time slot, a task is affecting the "cooperative" behavior requisite of the OSAL Task design.

```
GenericApp_NwkState = DEV_INIT;
```

It is useful to maintain a local copy of the device's network state. The network state at power-up is "not connected" or DEV_INIT. An OSAL task will not get a ZDO_STATE_CHANGE message of this default state during or after power-up, so it must be initialized locally. As soon as a new network state is achieved, the task will get the ZDO_STATE_CHANGE message. Note that when a device is built with NV_RESTORE and is connected to a network before a power cycle, the ZDO_STATE_CHANGE message will be received shortly after power-up with no OTA traffic because the "network connected" state has been restored from non-volatile memory.

```
GenericApp_DstAddr.addrMode = (afAddrMode_t)AddrNotPresent;
GenericApp_DstAddr.endPoint = 0;
GenericApp_DstAddr.addr.shortAddr = 0;
```

The default destination address is initialized so that messages will be sent as bound messages; using AddrNotPresent will force a binding table lookup for the destination address. If no matching binding exists, the message will be dropped.

```
GenericApp_epDesc.endPoint = GENERICAPP_ENDPOINT;
GenericApp_epDesc.task_id = &GenericApp_TaskID;
GenericApp_epDesc.simpleDesc =
    SimpleDescriptionFormat_t *)&GenericApp_SimpleDesc;
GenericApp_epDesc.latencyReq = noLatencyReqs;
// Register the endpoint description with the AF
afRegister( &GenericApp_epDesc );
```

The GenericApp Application Object is instantiated by the above code. This allows the AF layer to know how to route incoming packets destined for the profile/endpoint – it will do so by sending an OSAL SYS_EVENT_MSG-message (AF_INCOMING_MSG_CMD) to the task (task ID).

```
RegisterForKeys( GenericApp_TaskID );
```

The sample application registers for the exclusive system service of key press notification.[5]

1.6.2 Event Processing

Whenever an OSAL event occurs for the sample application, the event processing function (see 1.2.6.2 Task Event Handler), will be invoked in turn from the OSAL task processing loop. The parameter to the sample application task event handler is a 16-bit bit mask; one or more bits may be set in any invocation of the function. If more than one event is set, it is strongly recommended that a task should only act on one of the events (probably the most time-critical one, and almost always, the SYS_EVENT_MSG as the highest priority one).

```
if ( events & SYS_EVENT_MSG )
{
  MSGpkt = (afIncomingMSGPacket_t*)osal_msg_receive( GenericApp_TaskID );
  while ( MSGpkt )
  {
    …
```

Notice that although it is recommended that a task only act on one of possibly many pending events on any single invocation of the task processing function, it is also recommended (and implemented in the sample applications) to process all of the possibly many pending SYS_EVENT_MSG messages all in the same "time slice" from the OSAL.


```
switch ( MSGpkt->hdr.event )
```

Above it is shown how to look for the "type" of the SYS_EVENT_MSG message. It is recommended that a task implement a minimum subset of all of the possible types of SYS_EVENT_MSG messages.[6] This minimum subset is implemented by the GenericApp and described below.

```
case KEY_CHANGE:
  GenericApp_HandleKeys( ((keyChange_t *)MSGpkt)->state,
                         ((keyChange_t *)MSGpkt)->keys );
  break;
```

If an OSAL Task has registered for the key press notification, any key press event will be received as a KEY_CHANGE system event message. There are two possible paths of program flow that would result in a task receiving this KEY_CHANGE message.

The program flow that results from the physical key press is the following:

- HAL detects the key press state (either by an H/W interrupt or by H/W polling.)
- The HAL OSAL task detects a key state change and invokes the OSAL key change callback function.
- The OSAL key change callback function sends an OSAL system event message (KEY_CHANGE) to the Task Id that registered to receive key change event notification (RegisterForKeys().)

```
case AF_DATA_CONFIRM_CMD:
  // The status is of ZStatus_t type [defined in ZComDef.h]
  // The message fields are defined in AF.h
  afDataConfirm = (afDataConfirm_t *)MSGpkt;
  sentEP = afDataConfirm->endpoint;
  sentStatus = afDataConfirm->hdr.status;
  sentTransID = afDataConfirm->transID;
```

Any invocation of AF_DataRequest() that returns ZSuccess will result in a "callback" by way of the AF_DATA_CONFIRM_CMD system event message.

The sent Transaction Id (sentTransID) is one way to identify the message. Although the sample application will only use a single Transaction Id, it might be useful to keep a separate Transaction Id for each different endpoint or even for each cluster ID within an endpoint for the sake of message confirmation, retry, disassembling and reassembling, etc.. Note that any transaction ID state variable gets incremented by AF_DataRequest() upon success (thus it is a parameter passed by reference, not by value.)

The return value of AF_DataRequest() of ZSuccess signifies that the message has been accepted by the Network Layer which will attempt to send it to the MAC layer which will attempt to send it OTA. The sent Status (sentStatus) is the OTA result of the message. ZSuccess signifies that the message has been delivered to the next-hop ZigBee device in the network. If AF_DataRequest() was invoked with the AF_ACK_REQUEST flag, then ZSuccess signifies that the message was delivered to the destination address. Unless the addressing mode of the message was indirect (i.e. the message was sent to the network reflector to do a binding table lookup and resend the message to the matching device(s)), in which case ZSuccess signifies that the message was delivered to the network reflector. There are several possible sent status values to indicate failure.[7]

```
case ZDO_STATE_CHANGE:
  GenericApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
  if (  (GenericApp_NwkState == DEV_ZB_COORD)
      ||(GenericApp_NwkState == DEV_ROUTER)
      ||(GenericApp_NwkState == DEV_END_DEVICE) )
  {
    // Start sending "the" message in a regular interval.
    osal_start_timer( GENERICAPP_SEND_MSG_EVT,
                      GENERICAPP_SEND_MSG_TIMEOUT );
```



```
            }
            break;
```

Whenever the network state changes, all tasks are notified with the system event message ZDO_STATE_CHANGE.

```
        // Release the memory
        osal_msg_deallocate( (uint8 *)MSGpkt );
```

Notice that the design of the OSAL messaging system requires that the receiving task re-cycle the dynamic memory allocated for the message. If OSAL cannot enqueue a message (either the Task Id does not exist or the message header is not correct), it will re-cycle the memory.

The task processing function should try to get the next pending SYS_EVENT_MSG message:

```
        // Get the next message
        MSGpkt = (afIncomingMSGPacket_t*)osal_msg_receive(GenericApp_TaskID);
      }
```

After processing the SYS_EVENT_MSG messages, the processing function should return the unprocessed events:

```
      // Return unprocessed events
      return ( events ^ SYS_EVENT_MSG);
    }
```

1.7 Message Flow

To send a message, the sample applications call AF_DataRequest():

```
void GenericApp_SendTheMessage( void )
{
  char theMessageData[] = "Hello World";

  if ( AF_DataRequest( &GenericApp_DstAddr, &GenericApp_epDesc,
                       GENERICAPP_CLUSTERID,
                       (byte)osal_strlen( theMessageData ) + 1,
                       (byte *)&theMessageData,
                       &GenericApp_TransID,
                       AF_DISCV_ROUTE,
                       AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
  {
    // Successfully requested to be sent.
  }
  else
  {
    // Error occurred in request to send.
  }
}
```

When an Application Object makes a request to the ZDO to perform an auto match (reference “Auto Find” section), the network address and the endpoint number of the match can be save or filtered upon receipt. Although the code shows that the address mode can be taken from the response, it is important to realize that the only possible address mode of said response is afAddr16Bit per the ZigBee specification.

The call to AF_DataRequest() starts the process of passing the user message down through the layers of the stack in order to prepare a message to go OTA. If the function returns ZSuccess, then a complete message has been assembled to the point of having the network layer headers, footers, and optional security applied to the entire message, and this assembled message has been enqueued in the network buffers waiting to send OTA. The network layer will not attempt to pass this new message to the MAC until the Network Task runs again when it is invoked in order by the OSAL Task processing loop. Even when the Network Task runs again, the new OTA message will wait until any previously enqueued messages have been sent to the MAC to go OTA. Any return value of failure by AF_DataRequest() signifies a failure at one of the stack layers and is almost always due to a lack of sufficient heap space to enqueue another message; thus, there is no chance that the message went OTA.

When the Network Layer successfully passes the message to the MAC Layer and the MAC layer succeeds in sending the message OTA, the message is routed, hop by hop, to the destination address specified in the call to AF_DataRequest(). When the message finally arrives at the destination Network Address, the lower layers strip off optional security and route the Application Object data payload to the destination endpoint specified in the destination Address of passed as the first parameter in the call to AF_DataRequest(). The receiving Application Object will be notified by the SYS_EVENT_MSG message AF_INCOMING_MSG_CMD.

```
case AF_INCOMING_MSG_CMD:
  GenericApp_MessageMSGCB( MSGpkt );
  break;
```

Above, the sample application receives the SYS_EVENT_MSG message in task event handler and below, processes the user data sent OTA.

```
void GenericApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
  switch ( pkt->clusterId )
  {
    case GENERICAPP_CLUSTERID:
      // Put your code here to process the incoming data
      break;
  }
}
```


2. The GenericApp Sample Application

2.1 Introduction

The GenericApp provides a simple example of the structure of an application and the program and message flow. It will be illustrative to study the program and message flow in such a simple application. This application will send a simple message every 5 seconds. The message, "Hello World", is sent over-the-air to another device (Hand Bind or Auto Find). The message can only be seen with a packet sniffer. This application isn't intended as a "demo" application (it doesn't really do anything useful), but is intended to be a user's starting point to build their own application.

2.2 Messages

Notice that this sample application is implemented to begin a running timer as soon as the device successfully joins a network. Although the message isn't sent from the device until the destination address or a binding is set up.

```
if ( events & GENERICAPP_SEND_MSG_EVT )
{
  // Send "the" message
  GenericApp_SendTheMessage();

  // Setup to send message again
  osal_start_timer( GENERICAPP_SEND_MSG_EVT,
                    GENERICAPP_SEND_MSG_TIMEOUT );

  // return unprocessed events
  return (events ^ GENERICAPP_SEND_MSG_EVT);
}
```

The GenericApp has defined one of the 15 available bits in the Task Event Mask in `GenericApp.h`:[8]

```
#define GENERICAPP_SEND_MSG_EVT       0x0001
```

GenericApp uses its own task Id implicitly in the call to `osal_start_timer()` when setting a timer for itself for its `GENERICAPP_SEND_MSG_EVT` event. The timer was automatically started after receiving notification of successfully joining a network (`ZDO_STATE_CHANGE`). Above, the timer is re-started after every expiration. Every `GENERICAPP_SEND_MSG_TIMEOUT` interval, there is also a data request made in `GenericApp_SendTheMessage();`

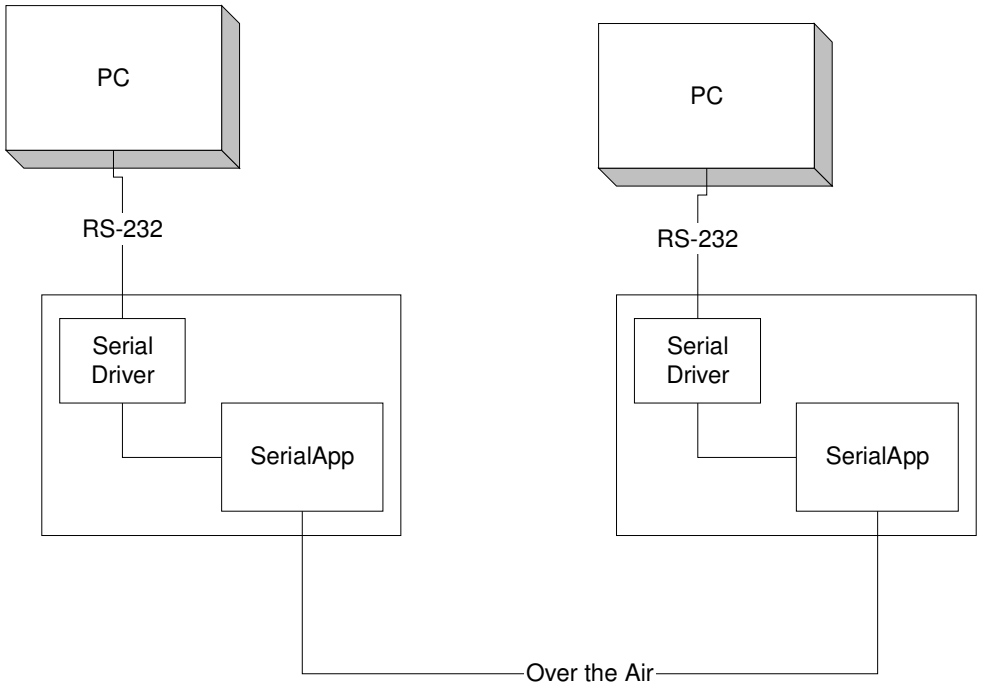
2.3 Key Presses

- SW1:
- SW2: Hand Binding by ZDP_EndDeviceBindReq().
- SW3:
- SW4: Auto Find by ZDP_MatchDescReq().

3. SerialApp The Sample Application

3.1 Introduction

SerialApp is basically a cable replacement for two non-ZigBee devices. A PC (or any non-ZigBee device) transfers data on a serial port connected to a ZigBee device running this application. This application then transmits the data stream from the serial port OTA to another ZigBee device also running SerialApp (device discovery and binding must have been completed between the two SerialApp Objects). The ZigBee Device that receives the SerialApp OTA data stream then transmits the data on its own serial port. The serial data transfer is designed to be bi-directional and full-duplex with hardware flow control enforced to allow for the OTA (multi-hop) delivery time and lost data packet retries.



3.2 Serial Setup

This application controls the serial directly. The driver setup is accomplished with the following in the task's initialization function:

```
uartConfig.configured           = TRUE;              // 2430 don't care.
uartConfig.baudRate             = SERIAL_APP_BAUD;
uartConfig.flowControl          = TRUE;
uartConfig.flowControlThreshold = SERIAL_APP_THRESH;
uartConfig.rx.maxBufSize        = SERIAL_APP_RX_MAX;
uartConfig.tx.maxBufSize        = SERIAL_APP_TX_MAX;
uartConfig.idleTimeout          = SERIAL_APP_IDLE;   // 2430 don't care.
uartConfig.intEnable            = TRUE;              // 2430 don't care.
uartConfig.callBackFunc         = rxCB;
HalUARTOpen (SERIAL_APP_PORT, &uartConfig);
```


3.3 Incoming Serial Data

The program flow that results from the serial driver sending a message is this:[9]

- HAL receives incoming serial data (either by H/W Rx interrupt or by Rx polling.)
- HAL driver reads and processes the received serial data until either the flowControlThresold or the idleTimeout is reached [both setup in the call to HalUARTOpen()], then it calls the receive callback function [rxCB()] to process the incoming serial data.

3.4 Outgoing Serial Data

Data is sent out the serial port by calling HalUARTWrite().

3.5 Key Presses

- SW1:
- SW2: Hand Binding by ZDP_EndDeviceBindReq().
- SW3:
- SW4: Auto Find by ZDP_MatchDescReq().

4. The TransmitApp Sample Application

4.1 Introduction

The TransmitApp application is used to demonstrate maximum throughput OTA by sending data packets as fast as the stack confirms the last data packet sent. The application will calculate (and display if built with LCD or serial port support) both the running average number of bytes per second and the total number of bytes for both transmitted and received packets.

4.2 Key Presses

- SW1: TransmitApp toggles the OTA transmission of messages.
- SW2: Hand Binding by ZDP_EndDeviceBindReq().
- SW3: TransmittApp clears the displayed metrics.
- SW4: Auto Find by ZDP_MatchDescReq().

4.3 Constants

The following constants are defined to put different variations on the messages:

- TRANSMITAPP_RANDOM_LEN – This enables the transmission of packets of randomized lengths. Otherwise, all transmitted packets will be the maximum length allowed by the AF layer for the given network configuration (e.g. security, protocol version, etc.)
- TRANSMITAPP_DELAY_SEND – If defined there will be an added delay between outgoing messages.
- TRANSMITAPP_SEND_DELAY – If TRANSMITAPP_DELAY_SEND is defined, this value is the amount of additional delay.
- TRANSMITAPP_TX_OPTIONS – This defines the TX options for the outgoing packet. If this item is defined as AF_MSG_ACK_REQUEST, outgoing packets will have APS ACK enabled and the receiving device will acknowledge that the packet was delivered. If APS ACK is enabled the transmit rate will decrease for the extra acknowledge message.

4.4 Metrics

In order to see the running metrics of the OTA traffic, the device must either have an LCD or be connected to ZTrace or ZTool.

The ZTool/ZTrace log window will display these parameters[10]:

- Parameter 1: Running average of the number of bytes being received per second.
- Parameter 2: Accumulated count of the total number of bytes received.
- Parameter 3: Running average of the number of bytes being transmitted per second.
- Parameter 4: Accumulated count of the total number of bytes transmitted.

If there is an LCD:

- Line 1: Parameters 1 & 2.
- Line 2: Parameters 3 & 4.

5. The Home Automation Profile

Whereas the previous sample applications have implemented fictitious Private Profiles, the HA Profile, Profile Id 0x0104, is a Stack Profile.

5.1 Introduction

The HA Profile relies upon the ZCL in general and specifically on the Foundation and General function domain.

5.2 Initialization

As described in section 1.2.2, the user must add at least one task (to the OSAL Task System) to service the one or many Application Objects instantiated. But when implementing an HA Application Object, a task dedicated to the ZCL must also be added (see the "OSAL Tasks" section) in the order shown in the HA Sample Applications (i.e. before any other tasks using ZCL).

In addition to the user task initialization procedure studied in the GenericApp,[11] the initialization of a HA Application Object also requires steps to initialize the ZCL. The following HA Sample Applications (Light and Switch) implement this additional initialization in the task initialization function (e.g. `zclSampleLight_Init()`)as follows.

- Register the *simple descriptor* (e.g. zclSampleLight_SimpleDesc) with the HA Profile using `zclHA_Init ()`.
- Register the *command callbacks table* (e.g. `zclSampleLight_CmdCallbacks)` with the General functional domain using `zclGeneral_RegisterCmdCallbacks()`.
- Register the *attribute list* (e.g. `zclSampleLight_Attrs)` with the ZCL Foundation layer using `zcl_registerAttrList()`.

5.3 Sample Light Application

5.3.1 Introduction

This sample application can be used to turn on/off the LED 4 on the device using the On/Off cluster commands, or put the device into the Identification mode (i.e., blinking the LED 4) by setting the IdentifyTime attribute to a non-zero value using the ZCL Write command.

5.3.2 ZCL Cluster Attributes

The Sample Light Application supports the following ZCL cluster attributes:

- The Basic cluster attributes:
  - zclSampleLight_HWRevision
  - zclSampleLight_ZCLVersion
  - zclSampleLight_ManufacturerName
  - zclSampleLight_ModelId
  - zclSampleLight_DateCode
  - zclSampleLight_PowerSource
  - zclSampleLight_LocationDescription
  - zclSampleLight_PhysicalEnvironment
  - zclSampleLight_DeviceEnable

- The Identify cluster attributes:
  - zclSampleLight_IdentifyTime
- The Scene cluster attributes:
  - zclSampleLight_SceneCount
  - zclSampleLight_CurrentScene
  - zclSampleLight_CurrentGroup
  - zclSampleLight_SceneValid
- The On/Off cluster attributes:
  - zclSampleLight_OnOff
- The On/Off switch Configuration cluster attributes:
  - zclSampleLight_SwitchType
  - zclSampleLight_SwitchActions
- The Level Control cluster attributes:
  - zclSampleLight_CurrentLevel
  - zclSampleLight_TransitionTime

5.3.3 ZCL Callback Functions

This application provides the following callback functions to the ZCL clusters:

- `zclSampleLight_BasicResetCB()` – called when a Reset to Factory Defaults command is received by the Basic cluster .
- `zclSampleLight_IdentifyCB()` - called when an Identify command is received by the Identify cluster, which in turn calls `zclSampleLight_ProcessIdentifyTimeChange()` to process any changes to the value of the Identify Time attribute. This function puts the device into the Identification mode if the value of the Identify Time attribute is non-zero. Otherwise, it puts the device back into its normal operational mode. While in the Identification mode, the device puts its LED 4 into the blinking state and decrements the Identify Time attribute every second.
- `zclSampleLight_IdentifyQueryRspCB()` - called when an Identify Query Response is received by the Identify cluster.
- `zclSampleLight_OnOffCB()` – called when an On, Off or Toggle command is received by the On/Off cluster .

5.3.4 Key Presses

- SW1: No action defined.
- SW2: Hand Binding by ZDP_EndDeviceBindReq().
- SW3: No action defined.
- SW4: No action defined.

5.4 Sample Switch Application

5.4.1 Introduction

This sample application can be used as the Light Switch (using the SW 1) to turn on/off the LED 4 on a device running the Sample Light application.

5.4.2 ZCL Cluster Attributes

The Sample Switch Application supports the following ZCL cluster attributes:

- The Basic cluster attributes:
  - zclSampleSw_HWRevision
  - zclSampleSw_ZCLVersion
  - zclSampleSw_ManufacturerName
  - zclSampleSw_ModelId
  - zclSampleSw_DateCode
  - zclSampleSw_PowerSource
  - zclSampleSw_LocationDescription
  - zclSampleSw_PhysicalEnvironment
  - zclSampleSw_DeviceEnable
- The Identify cluster attributes:
  - zclSampleSw_IdentifyTime
- The On/Off cluster attributes:
  - zclSampleSw_OnOff

5.4.3 ZCL Callback Functions

This application provides the following callback functions to the ZCL clusters:

- `zclSampleSw_BasicResetCB()` – called when a Reset to Factory Defaults command is received by the Basic cluster.
- `zclSampleLight_IdentifyCB()` - called when an Identify command is received by the Identify cluster, which in turn calls `zclSampleLight_ProcessIdentifyTimeChange()` to process any changes to the value of the Identify Time attribute. This function puts the device into the Identification mode if the value of the Identify Time attribute is non-zero. Otherwise, it puts the device back into its normal operational mode. While in the Identification mode, the device puts its LED 4 into the blinking state and decrements the Identify Time attribute every second.
- `zclSampleSw_IdentifyQueryRspCB()` - called when an Identify Query Response is received by the Identify cluster.

5.4.4 Key Presses

- SW1: Send a Toggle message to the Light.
- SW2: Hand Binding byZDP_EndDeviceBindReq().
- SW3: No action currently defined.
- SW4: Auto Find by ZDP_MatchDescReq().

Appendix A. Applicable Documents

1. Z-Stack OSAL API , SWRA194
2. Z-Stack API, SWRA195
3. Z-Stack ZCL API, SWRA197
4. Z-Stack Compile Options, SWRA188

[1] If your application will not be implementing a Standard or a Published Profile, then you must apply for a Private Profile Id that can be used outside of the development laboratory. The process of obtaining a Private Profile Id is outside the scope of this document

[2] The changes required to instantiate more than one sample application in the same device are discussed in the document where the corresponding functionality is described.

[3] The next hop device may or may not be the destination device of a data request. Therefore, the AF data confirmation of delivery to the next hop must not be misinterpreted as confirmation that the data request was delivered to the destination device. Refer to the discussion of delivery options (`AF_ACK_REQUEST` in the Z-Stack API).

[4] See sections 1.2.1 and 1.2.5.1.

[5] See sections 1.2.3 and 1.2.6.1.3.

[6] For a complete list of all of the types of SYS_EVENT_MSG messages, refer to "Global System Messages" in ZComDef.h.

[7] See "MAC status values" in ZComDef.h.

[8] Note that one task event is reserved, see "Global System Events" in ZComDef.h:

```
#define SYS_EVENT_MSG               0x8000  // A message is waiting event
```

[9] In order to receive Z-Tool messages, the device must be built with MT_TASK defined, as well as the other related options required to set-up the serial port for MT – see the Z-Stack Compile Options document.

[10] Note that packing and sending messages OTA is asynchronous with receiving and parsing them. Therefore, the average bytes per second displayed by the transmitting device may vary by several packet sizes per second from that displayed by the receiving device. Both rates may jitter and are expected to slow measurably if there is significant other channel traffic, if the transmit is run bi-directional, or if there is significant channel interference from non-ZigBee devices. If an end device is used, the transmit rate will be greatly reduced because of the end device's polling rate.

