



Z-Stack
Sample Application
For SmartRF05 and CC2530

Document Number: SWRA232

Texas Instruments, Inc.
San Diego, California USA

Version	Description	Date
1.0	Initial release.	09/08/2008
1.1	Updated for ZStack 2.2.0 release	04/02/2009

TABLE OF CONTENTS

I. ACRONYMS.....	III
1. Z-STACK CC2530EB SAMPLE APPLICATION	1
1.1 INTRODUCTION	1
1.1.1 Description	1
1.1.2 User Application Development.....	1
2. THE SAMPLE APPLICATION (SAMPLEAPP).....	2
2.1 INTRODUCTION	2
2.2 PROGRAM FLOW	2
2.2.1 Initialization.....	2
2.2.2 Event Processing	3
2.3 MESSAGE FLOW.....	4
3. DEMO BUILD	6
3.1 DEMO JUMPER SETTING.....	6

TABLE OF FIGURES

FIGURE 1: JUMPERS SETTING ON SMARTRF05 – REV 1.3	6
FIGURE 2: JUMPERS SETTING ON SMARTRF05 – REV 1.7	6

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 CC2530EB Sample Application

1.1 Introduction

This document covers the Z-Stack™ Sample Application for the Texas Instruments SmartRF05EB and CC2530EM boards, also known as the CC2530EB. This sample application is a simple head-start to using the TI distribution of the ZigBee Stack in order to implement a specific Application Object.

1.1.1 Description

This sample application is a very simple demo application for the ZStack CC2530 Developer Boards.

Each application can send and receive 2 messages:

- Periodic Message – Each device will send this broadcast message every 10 seconds (plus a small random number of milliseconds for jitter) after joining the network. The message's payload is the sending device's periodic message counter.
- Flash Control Message – This message contains a counter (number of times this message has been sent from a device) and the number of milliseconds to flash the LED (in milliseconds). This message is sent when SW1 is pressed. This message is sent to APS Group 1 (this is a broadcast message that is filtered for devices that belong to group 1).

All devices, during initialization, join group 1, then the user can remove the application from group 1 by pressing SW2. SW2 works like a toggle to join and remove the device from group 1. So, by pressing SW2 the user can disable the LED flashing (not accept Flash Control Messages).

1.1.1.1 Key Presses

- SW1: Sends a flash message to all devices in group 1.
- SW2: Toggles the device's enrollment in and out of group 1.

1.1.2 User Application Development

Before continuing, please read section 1 of the Z-Stack Applications SWRA201 document.

This 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. In addition, 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, Z-Stack 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 numerical values 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

2. The Sample Application (SampleApp)

2.1 Introduction

The SampleApp provides an 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.

2.2 Program Flow

2.2.1 Initialization

During system power-up and initialization, SampleApp_Init() will be invoked.

```
SampleApp_TaskID = task_id;
```

Notice that OSAL assigns the SampleApp its Task Id via the function parameter. This is the Task Id that SampleApp 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 effecting the “cooperative” behavior requisite of the OSAL Task design.

```
SampleApp_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.

```
SampleApp_DstAddr.addrMode = (afAddrMode_t)AddrNotPresent;
SampleApp_DstAddr.endPoint = 0;
SampleApp_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.

```
SampleApp_epDesc.endPoint = SAMPLEAPP_ENDPOINT;
SampleApp_epDesc.task_id = &SampleApp_TaskID;
SampleApp_epDesc.simpleDesc =
    SimpleDescriptionFormat_t *)&SampleApp_SimpleDesc;
SampleApp_epDesc.latencyReq = noLatencyReqs;
// Register the endpoint description with the AF
afRegister( &SampleApp_epDesc );
```

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

```
RegisterForKeys( SampleApp_TaskID );
```

The SampleApp registers for the exclusive system service of key press notification.

```
SampleApp_Group.ID = 0x0001;
osal_memcpy( SampleApp_Group.name, "Group1" );
aps_AddGroup( SAMPLEAPP_ENDPOINT, &SampleApp_Group );
```

Flash Control messages are sent to group 1, so to receive messages for group 1, this device will be automatically setup for group 1. Pressing SW1, will toggle the enrollment into group 1.

2.2.2 Event Processing

Whenever an OSAL event occurs for the SampleApp_TaskID, the SampleApp event processing function, SampleApp_ProcessEvent(), will be invoked in turn from the OSAL round-robin task processing loop. The parameter to SampleApp_ProcessEvent() 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( SampleApp_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.³ This minimum subset is implemented by the SampleApp and described below.

```
case KEY_CHANGE:
    SampleApp_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 SampleApp only uses a single Transaction Id (SampleApp_TransID), it might be useful to keep a separate Transaction Id for each different EndPoint or even for each ClusterId within an EndPoint for the sake of message confirmation, retry, disassembly, and reassembly, 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.⁴

```

case ZDO_STATE_CHANGE:
    SampleApp_NwkState = (devStates_t)(MSGpkt->hdr.status);
    if ( (SampleApp_NwkState == DEV_ZB_COORD)
        || (SampleApp_NwkState == DEV_ROUTER)
        || (SampleApp_NwkState == DEV_END_DEVICE) )
    {
        // Update the LCD's network indicator

        // Start sending "the" message in a regular interval.
        osal_start_timer( SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
                        SAMPLEAPP_SEND_PERIODIC_MSG_TIMEOUT );
    }
    break;

```

Whenever the network state changes, all tasks are notified with the system event message `ZDO_STATE_CHANGE`. Notice that `SampleApp` is implemented to begin a running timer as soon as the device successfully joins a network. When the network state changes to “joined”, it might also be useful to initiate an auto find so that devices can bind without requiring user action.

```

// 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.

```

if ( events & SAMPLEAPP_SEND_PERIODIC_MSG_EVT )
{
    // Send "the" message
    SampleApp_SendPeriodicMessage();

    // Setup to send message again
    osal_start_timer( SAMPLEAPP_SEND_PERIODIC_MSG_EVT,
                    SAMPLEAPP_SEND_MSG_TIMEOUT );

    // return unprocessed events
    return (events ^ SAMPLEAPP_SEND_PERIODIC_MSG_EVT);
}

```

The `SampleApp` has defined one of the 15 available bits in the Task Event Mask in `SampleApp.h`.⁵

```
#define SAMPLEAPP_SEND_PERIODIC_MSG_EVT    0x0001
```

`SampleApp` uses its own Task Id implicitly in the call to `osal_start_timer()` when setting a timer for itself for its `SAMPLEAPP_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 `SAMPLEAPP_SEND_PERIODIC_MSG_EVT` interval, there is also a data request made in `SampleApp_SendPeriodicMessage()`;

2.3 Message Flow

By using an OSAL timer, `SampleApp` sends a periodic message OTA.

```

void SampleApp_SendPeriodicMessage( void )
{

```

```

afAddrType_t dstAddr;

dstAddr.addrMode = afAddrBroadcast;
dstAddr.addr.shortAddr = 0xFFFF; // Broadcast to everyone
dstAddr.endpoint = SAMPLEAPP_ENDPOINT;

if ( AF_DataRequest( & dstAddr, &SampleApp_epDesc,
                    SAMPLEAPP_PERIODIC_CLUSTERID,
                    (uint8)sampleAppPeriodicCounter++,
                    (uint8 *)&sampleAppPeriodCounter,
                    &SampleApp_TransID, AF_DISCV_ROUTE,
                    AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
{
    // Successfully requested to be sent.
}
else
{
    // Error occurred in request to send.
}
}

```

The Application Object data of the OTA message is a simple counter, that when received will be displayed on the receiver's LCD.

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:
    SampleApp_MessageMSGCB( MSGpkt );
    break;

```

Above, the `SampleApp` receives the `SYS_EVENT_MSG` message in `SampleApp_ProcessEvent()` and below, processes the user data sent OTA.

```

void SampleApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
    switch ( pkt->clusterId )
    {
        case SAMPLEAPP_PERIODIC_CLUSTERID:
            // Display and increment a counter on the LCD in the periodic space
            break;
        case SAMPLEAPP_FLASH_CLUSTERID:
            flashTime = BUILD_UINT16(pkt->cmd.Data[1], pkt->cmd.Data[2] );
            HalLedBlink( HAL_LED_4, 4, 50, (flashTime / 4) );
            break;
    }
}

```

The contents of the OTA data transfer are not used. SampleApp assumes that the OTA data contains a counter and a flash timeout (for the flash control message).

3. Demo Build

Along with the usual ZigBee device configurations (Coordinator, Router and End-Device), this project includes a “Demo” configuration. The “Demo” configuration(s) can become either a coordinator or router (from the same build) depending on a hardware jumper setting. So, you can load the same (“Demo”) build on to all of your demo kit boards, then set a jumper on one of the boards (to become the coordinator) before starting all the devices to form a network.

3.1 Demo Jumper Setting

The “Coordinator” jumper for this application is located on the developer’s boards:

Board	Jumper Location
SmartRF05 (Rev 1.3) + CC2530	Port 18 pin 7 & 9
SmartRF05 (Rev 1.7) + CC2530	Port 18 pin 9 & 11

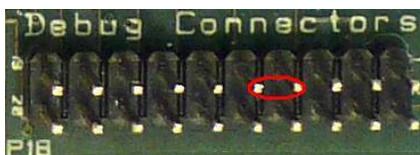


Figure 1: Jumpers Setting on SmartRF05 – Rev 1.3

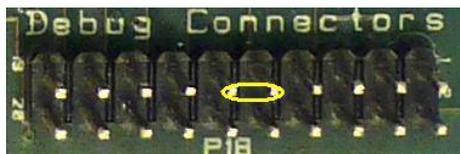


Figure 2: Jumpers Setting on SmartRF05 – Rev 1.7

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 For a complete list of all of the types of SYS_EVENT_MSG messages, refer to “Global System Messages” in ZComDef.h.

⁴ See “MAC status values” in ZComDef.h.

⁵ Note that one task event is reserved, see “Global System Events” in ZComDef.h:

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