Chipcon Products from Texas Instruments

Application Note: Migrating From Z-Stack Version 1.3 to 1.4

Document Number: F8W-2006-0004

Texas Instruments, Inc.
San Diego, California USA
(619) 497-3845

Version	Description	Date
1.0	Initial release.	12/08/2006
1.1	Changed document name. Updated title page.	05/21/2007

TABLE OF CONTENTS

Purpose

This document describes the considerations one needs to take into account when migrating from the 1.3.x version of Z-Stack™ to the 1.4.0 version of Z-Stack™. Both versions run on the CC2420/Atmega and CC2430 platforms.

1. Definitions

The following terms are used in this document:

MAC – Media Access Control software that implements the IEEE 802.15.4 specified communication functions.

IDE – Integrated Development Environment

OSAL – Operating System Abstraction Layer

MTEL – Monitor Test Event Loop

HAL – Hardware Abstraction Layer

API – Application Programming Interface

2. Software System Changes

The changes made to the software system in version 1.4.0 are summarized as follows:

1. System-level abstraction files. These files replace the functionality currently in ZComDef.h, OnBoard.h, and other platform-specific files.
2. HAL Drivers. These drivers provide a platform-independent API for common hardware services.
3. OSAL message service changes. The OSAL message service has been modified to allow an application to manage its own queue of OSAL message buffers.

3. System-Level Abstraction Files

The following new files are introduced in the software system for version 1.4.0:

hal_types.h
Location: Components\hal\target\*\hal_types.h[1]

This file contains the type definitions for system-wide common data types such as uint8, uint16, bool etc. It also contains memory attributes such as CODE and XDATA, plus system-wide defines for TRUE, FALSE, and NULL. There is a separate hal_types.h for each processor family.

hal_defs.h
Location: Components\hal\include\hal_defs.h

This file contains simple utility macros. This file is platform independent.

[1] (* represents the name of the specific target – e.g. CC2430EB)

hal_mcu.h
Location: Components\hal\target\*\hal_mcu.h

This file contains compiler abstractions for interrupts and I/O. It contains standardized interrupt vector names and macros for enabling and disabling interrupts. There is a separate hal_mcu.h for each processor family.

hal_board_cfg.h
Location: Components\hal\target\*\hal_board.h

This file contains board-level hardware abstractions for interrupts and I/O. It contains standardized macros for manipulating hardware such as switches and LEDs. It also contains non-standardized macros that are specific to the processor. **One should modify this file if the pin interfaces are different from the development kit hardware.**

hal_mac_cfg.h
Location: Components\hal\target\*\hal_mac_cfg.h

This file contains MCU-to-radio pin interface definitions. **One should modify this file if the pin interfaces are different from the development kit hardware.** For the CC2430 platform, this file does not exist since it is a single chip integrated solution that encompasses both radio and MCU.

4. HAL Drivers

The HAL drivers provide a platform-independent API to common hardware services. They consist of common interface files and separate implementation files for each target processor.

Location: Components\hal\include, Components\hal\target\*, Components\hal\common

Integration of the new HAL drivers required changes to OnBoard, ZMain, MTEL, MTSPCI, OSAL and the sample applications. It also affected any place that the ADC, KEYS, LCD, LED, TIMER and UART functions were used.

ADC

ADC/AtoD was removed from OnBoard.c. Anywhere ReadAtoD() and ReadAtoD8bit() got called was replaced with HalAdcRead().

KEYS

Key events and key handlers were removed from OnBoard.c. Keys are handled by the hal drivers task. Keys are now implemented as events sent back by a callback from hal_drivers. Previously, OnBoard.c trapped the key events and sent the keys to the application using the OSAL messaging system.

LCD

LCD functions were removed from OnBoard.c. LCD functions can now be called directly from hal_lcd.c

LED

LED functions were removed from OnBoard.c. LED functions can now be called directly from hal_led.c

TIMERS

Timers were removed from OnBoard.c. All the initialization and settings are done in hal_timer.c. On the CC2430 platform, OSAL_timers.c uses Timer 4 (8-bit) as a way to update its tick every 1ms. On the CC2420+AVR platform, OSAL_timers.c uses either Timer 0 (8-bit) or Timer 3 (16-bit) as a way to update its tick every 1ms, and one can configure which timer OSAL uses in the f8wConfig.cfg file. OSAL_timers implements a callback function which calls osal_update_timers() to update the tick.

On the CC2430 platform, the MAC uses Timer 2, and on the CC2420+AVR platform, the MAC uses Timer 1.

UART

UART functions were removed from OnBoard.c. The UART is initialized and updated through the hal_drivers task. UART callbacks are implemented in SPIMgr.c to trap and process Rx and Tx data. SPIMgr.c replaces MTSPCI.c.

MTEL.c and MTSPCI.c

MTEL.c was changed to adapt to the new HAL UART interface. MTEL interfaces to HAL UART to send data and to SPIMgr.c to receive data. MTEL also uses various SPIMgr utility functions. MTEL now only processes formatted data (ZTool) and no longer processes raw data (ZApp).

MTSPCI.c was removed and replaced with SPIMgr.c. SPIMgr reads data from HAL UART and processes it as formatted data (ZTool) or raw data (ZApp). After data is processed it is sent as an OSAL message to MTEL.

OSAL_Timers.c

OSAL_timers.c was changed to work with the new hal_timer.c. OSAL_timers has a callback function that just updates osal_update_timers() every tick. The tick and timer can be setup in osalTimerInit().

C:\Chipcon\CC-1.0-1.3.x	C:\Texas Instruments\CC-1.4.0
Onboard.c functions	
ADC	hal_adc.c
Timers	hal_timer.c
UART	hal_uart.c
LED	hal_led.c
LCD	hal_lcd.c
KEYS	hal_keys.c
MTSPCI.c	SPIMgr.c

Table 1. Mapping of old functions to new

5. OSAL Message Service Changes

The OSAL message service was modified to allow an application to manage its own queue of OSAL messages. This allows for more efficient handling of data queues and message passing in the MAC or any protocol layer.

Design

When an OSAL message is allocated, a hidden internal header is created in the allocated message. This header contains a pointer to the next item in the queue. This internal queueing mechanism was modified such that the pointers point to the user-accessible portion of the message rather than the internal header, as shown below.

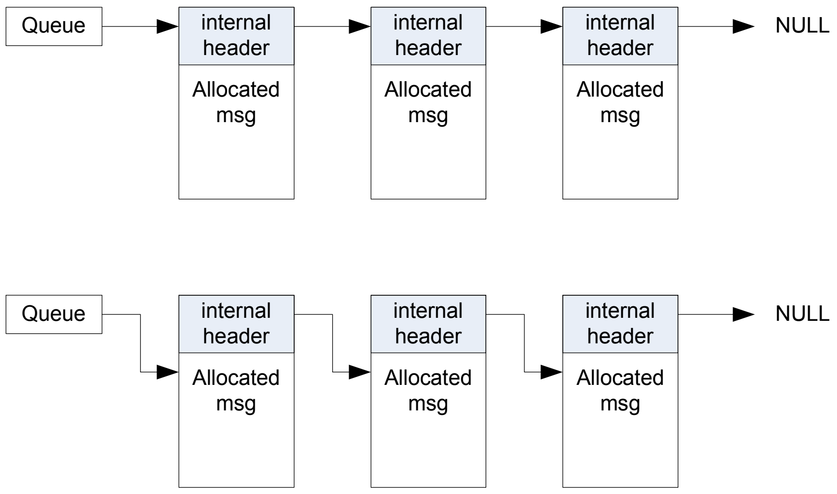

Figure 1. Modification of Internal Queuing Mechanism

This change allows an application to traverse a queue while hiding the internal header.

In addition, the following new macros and functions were added to OSAL.c and OSAL.h:

New macros:

- OSAL_MSG_Q_INIT(q_ptr): Initialize a message queue.
- OSAL_MSG_Q_EMPTY(q_ptr): Check if a queue is empty.
- OSAL_MSG_Q_HEAD(q_ptr): Get a pointer to the head of the queue.
- OSAL_MSG_NEXT(msg_ptr): Get message's pointer to the next message in the queue.

New functions:

- extern void osal_msg_enqueue(osal_msg_q_t *q_ptr, void *msg_ptr): Enqueue a task message.
- extern void osal_msg_enqueue_max(osal_msg_q_t *q_ptr, void *msg_ptr, byte max): Enqueue a message if queue length is less than max.
- extern void *osal_msg_dequeue(osal_msg_q_t *q_ptr): Dequeue a task message.
- extern void osal_msg_push(osal_msg_q_t *q_ptr, void *msg_ptr): Push a message to head of the queue.
- extern void osal_msg_extract(osal_msg_q_t *q_ptr, void *msg_ptr, void *prev_ptr): Extract and remove a task message from the queue.

Other Changes

Other changes to the OSAL message service were made to reduce code size:

- Removed the length parameter from osal_msg_send().
- Modified "byte osal_msg_allocate(byte **msg_ptr, byte len)" to "void *osal_msg_allocate(byte len)". The function return value contains a pointer to the message instead of the function argument.
- The function osal_msg_receive returns type "byte *" instead of "osal_msg_received_t*"

6. Application Framework and Interface Changes

This section provides an example of the types of changes that would be of interest to an application developer moving from the 1.3.x to 1.4.0 version of Z-Stack. Although this section gives an overview of what changed with the application framework and interfaces, one would typically start with a sample application from the 1.4.0 stack and then port over the application specific code to the new 1.4.0 based project. One would use the new sample application files as a start point, porting only the application specific code. The example was taken from GenericApp, but provides some basic changes that are applicable across a range of applications

6.1 OSAL_MyApp.c

6.1.1 Headers

The following was added:

```
/* HAL */
#include "hal_drivers.h"
```

6.1.2 Registered Tasks

```
osalTaskAdd( OnBoard_TaskInit, OnBoard_ProcessEvent, OSAL_TASK_PRIORITY_LOW );
```

was replaced with:

```
osalTaskAdd (Hal_Init, Hal_ProcessEvent, OSAL_TASK_PRIORITY_LOW);
```

6.2 MyApp.h

Nothing was changed.

6.3 MyApp.c

6.3.1 Headers:

The following was added:

```
/* HAL */
#include "hal_lcd.h"
#include "hal_led.h"
#include "hal_key.h"
#include "hal_uart.h"
```

6.3.2 Global Variables

Version 1.4.0 of Z-Stack uses 16 bit Cluster ID's to extend the number of supported Clusters. In accordance with this change the cId_t type has been added as a Cluster ID type.

The data type for the MyApp_ClusterList was modified from **byte** to **cId_t**

6.3.3 Local Functions

To reduce overhead and code size, KVP is no longer supported in ZigBee 1.1, and as a result, has been removed from the sample application projects in the 1.4.0 stack. Any messaging that utilizes KVP must now be converted to the MSG format.

What was removed: void MyApp_MessageKVPCB(afIncomingKVPPacket_t *pckt);

6.3.4 MyApp_ProcessEvent

Removed KVP related code

- KVP Data Types
- KVP Message Handler code in Switch statement

Some data structures for handling OSAL messages have changed in version 1.4.0 to optimize code and make it more legible. To accommodate these changes, the code that handles and sorts message types in MyApp_ProcessEvent has been modified to fit these data structures. The following code is used to handle messages in the new 1.4.0 stack.

```
afIncomingMSGPacket_t *MSGpkt;

if ( events & SYS_EVENT_MSG )
{
  MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( GenericApp_TaskID );
  while ( MSGpkt )
  {
    switch ( MSGpkt->hdr.event )

    {
```

As a result of the modified data structure, the call to MyApp_HandleKeys has also changed. The following code is now used:

```
MyApp_HandleKeys(((keyChange_t *)MSGpkt)->state,((keyChange_t *)MSGpkt)->keys);
```

For Binding, the ZDO_NEW_DSTADDR event has been slightly modified. A ZDO_NewDstAddr_t structure has been created that allows one to extract the destination address directly from the message using the following code:

```
// ZDO Destination Address Declaration

ZDO_NewDstAddr_t *ZDO_NewDstAddr;

// Extract Destination Address from Packet

ZDO_NewDstAddr = (ZDO_NewDstAddr_t *)MSGpkt;
dstEP = ZDO_NewDstAddr->dstAddrDstEP;
dstAddr = &ZDO_NewDstAddr->dstAddr;
MyApp_DstAddr.addrMode = (afAddrMode_t)dstAddr->addrMode;
MyApp_DstAddr.endPoint = dstEP;
MyApp_DstAddr.addr.shortAddr = dstAddr->addr.shortAddr;
```

As a result of the change of the message structure pointer, a few other lines of code were modified.

```
// Retrieve Network State

GenericApp_NwkState = (devStates_t)(MSGpkt->hdr.status);

// Release memory

osal_msg_deallocate( (uint8 *)MSGpkt );

// Next System Event

MSGpkt = (afIncomingMSGPacket_t *)osal_msg_receive( GenericApp_TaskID );
```

6.3.5 MyApp_SendTheMessage

The interface to AF for sending Application Data has been modified. This API change was made to simplify our interface and remove KVP related fields. The old afFillAndSendMessage function call has been replaced with the following function:

```
afStatus_t AF_DataRequest(
                    afAddrType_t *dstAddr,  // Packet Destination Address
                    endPointDesc_t *srcEP,  // Source Endpoint
                    uint16 cID,             // Destination Cluster ID
                    uint16 len,             // Payload Length
                    uint8 *buf,             // Payload Data Pointer
                    uint8 *transID,         // Transaction ID
                    uint8 options,          // TX Options
                    uint8 radius )          // Maximum Propagation Radius
```

6.3.6 HAL Changes

Replaced EVAL_SWx with HAL_KEY_SW_x

<See previous sections for additional HAL Changes>

Previously to control LEDs one called SetLed(). Now one must call HalLedSet().

6.3.7 LCD Changes

Replaced:

WriteLCDString(<Line#>, <string>);

With:

HalLcdWriteString(<string>, HAL_LCD_LINE_<Line#>);

<See Previous sections for additional LCD Changes>

6.3.8 Configuration File

The most common compiler command-line options used to define a Z-Stack device can now be done by editing a configuration file called f8wConfig.cfg. This file can be found in …\Projects\zstack\Tools\CC2430DB. "See AN 025 – Application Level Tuning of Z-Stack" for more details.

7. Feature Enhancements for Zigbee v1.1 Specification

The following is a discussion of feature enhancements that were made to the Z-stack to support the Zigbee v1.1 specification.

7.1 Binding

The Binding table is not kept on the coordinator (even though the Z-Stack still supports this). End devices and routers now support source binding by using the compile option REFLECTOR. When end device binding is performed, the devices will still send the end device bind request to the coordinator (and the coordinator will still determine a match on endpoint and cluster ID values), but now the binding entry is created on the devices themselves and not on the coordinator. Therefore, when using source binding it eliminates the need for sending indirect messages that are transmitted first to the coordinator and then to the final destination. The procedure for binding using the Match Descriptor Request remains the same.

7.2 Security

The ZDO Security Manager offers APIs that can be used to manage application level security policies (ZDSecMgr.c). There are two new APIs called ZDSecMgrUpdateNwkKey() and ZDSecMgrSwitchNwkKey() that will allow the use of 2 NWK security keys. ZDSecMgrUpdateNwkKey() allows the user to cause a new NWK key to be sent to all the devices to the network. One of the parameters to this function is the key sequence number. The user can designate the key sequence number of the active key to be 0, and the new key to be 1 for example when calling this function. . ZDSecMgrSwitchNwkKey() will send a broadcast message to let all devices know to begin using the new key.

7.3 Broadcast Messaging

More filtering based on the short address used in broadcast messages has been added.

0xFFFD goes to all devices on which macRxOnWhenIdle is TRUE. Therefore, this may or may not include certain end devices.
0xFFFC goes to all routers and the Coordinator.
0xFFF8-**0xFFFB** and **0xFFFE** are reserved.

7.4 Group Addressing

Group addressing support has been added. A group message is a broadcast message that is filtered for the application by the group ID. It is essentially a broadcast message to the entire network where it only gets delivered to application level of those who belong to the group.

There are 2 parts to the group addressing - sending to a group address and receiving group messages.

Sending to a group:

A device can send direct to a group. When sending a data message to a group the destination address' address mode will be set to afAddrGroup and the short address will contain the group number (ID).

A device can send an indirect message to a group address, if the group address is in the device's binding table (bind_req - source bound). So, it must be bound to a cluster ID - just like a normal binding record.

Receiving a group message:

Group messages are sent to all devices as broadcast messages with a group ID in the APS frame. For the device to accept the group message, the group ID must be in the device's group table. If the message is accepted (match found in the group table), the message is sent to the endpoint from the group table.

To add an endpoint to the group table, the application must add it to the table by calling AddGroup().

Sample code for adding an endpoint to the group table:

```
void TransmitApp_HandleKeys( byte shift, byte keys ) {
  aps_Group_t group;
  // Shift is used to make each button/switch dual purpose.
  if ( shift )
  {
    if ( keys & HAL_KEY_SW_1 )
    {
      // Assign yourself to group 1
      group.ID = 0x0001;
      group.name[0] = 0;
      aps_AddGroup( TRANSMITAPP_ENDPOINT, &group );
    }
    if ( keys & HAL_KEY_SW_2 )
    {
      // Change destination address to group 1
      TransmitApp_DstAddr.addrMode = afAddrGroup;
      TransmitApp_DstAddr.endPoint = 0;
      TransmitApp_DstAddr.addr.shortAddr = 0x001; // group address
    }
```

The binding table entry structure has been modified to support the group address.