Introduction

This module will take a fairly comprehensive look at the ZigBee stack. We’ll delve into forming a network, descriptors, the application framework, binding and other topics. By the end of the module with its 4 labs you should be able to ask the right questions (and maybe answer them) concerning how your application will utilize the ZigBee stack.

Objectives

- Starting a network
- Routing
- Packet Sniffer
- Descriptors
- Application Framework
- Binding
- ZDO API’s
- Callbacks
- Multiple Endpoints
- Mobility
*** I was forced at gunpoint to leave this page blank ***
Module Topics

ZigBee Stack .............................................................................................................................................. 8-1

Module Topics........................................................................................................................................... 8-3
ZigBee Flavors........................................................................................................................................... 8-5
ZigBee 2007 Features.............................................................................................................................. 8-7
ZigBee 2007 Summary.............................................................................................................................. 8-10
ZigBee Advantages and Architecture................................................................................................. 8-11
Mesh Networking and Components................................................................................................. 8-12
PAN Formation........................................................................................................................................ 8-14
Address Allocation............................................................................................................................. 8-15
Routing.................................................................................................................................................. 8-16
Configuration......................................................................................................................................... 8-18
Lab 8A – Starting a PAN ..................................................................................................................... 8-19
  Description: ........................................................................................................................................ 8-19
  Hardware list: ................................................................................................................................... 8-20
  Software list: .................................................................................................................................... 8-20
  Procedure......................................................................................................................................... 8-21
  Send Data ....................................................................................................................................... 8-24
Profiles................................................................................................................................................ 8-25
Clusters............................................................................................................................................... 8-26
Endpoints .......................................................................................................................................... 8-27
Descriptors........................................................................................................................................ 8-28
Architecture ....................................................................................................................................... 8-31
Application Framework and Send/Receive Data .............................................................................. 8-32
Visualizing Acknowledgements......................................................................................................... 8-34
Third Party Tools.............................................................................................................................. 8-35
Writing to the LCD ............................................................................................................................ 8-39
Lab 8B – Code Walkthrough............................................................................................................... 8-41
  Description: ........................................................................................................................................ 8-41
  Hardware list: ................................................................................................................................... 8-42
  Software list: .................................................................................................................................... 8-42
  Procedure......................................................................................................................................... 8-43
  Application Level Acknowledgements............................................................................................. 8-46
Digging Further into the Stack ............................................................................................................ 8-48
Binding............................................................................................................................................... 8-49
Z-Tool ................................................................................................................................................. 8-53
Lab 8C – Clusters, Binding and Z-Tool ............................................................................................... 8-57
  Hardware list: ................................................................................................................................... 8-58
  Software list: .................................................................................................................................... 8-58
Module Topics

Procedure...................................................................................................................... 8-59

Changing the reporting period .................................................................................. 8-60
  Adding a Control Cluster ...................................................................................... 8-62
  Using Centralized Binding .................................................................................. 8-65
  Z-Tool ................................................................................................................... 8-67

ZDO ............................................................................................................................ 8-69

Mobility....................................................................................................................... 8-71

Lab 8D – Send and Receive Control Cluster ............................................................. 8-73
  Hardware list: ........................................................................................................ 8-74
  Software list: ......................................................................................................... 8-74

Procedure...................................................................................................................... 8-75
  Message Timing .................................................................................................. 8-78

Smart Energy ............................................................................................................. 8-80

Demos ......................................................................................................................... 8-83
ZigBee Flavors

ZigBee Snapshot - Sept 2008

- IEEE 802.15.4-2003 defines PHY/MAC
- ZigBee 2006
  - Products shipping today
- ZigBee 2007
  - Two stack profiles: ZigBee and ZigBee PRO

ZigBee 2006

- Extremely well tested by a variety of companies
- Base of products and networks on the market and in use today
- Many certified stacks and silicon providers available
- Simpler: Less code and overhead than 2007 or PRO
**ZigBee 2007 and ZigBee PRO**

- **ZigBee 2007**
  - Based on proven 2006 feature set plus frequency agility and optional fragmentation
  - Basic features require less memory & resources than PRO

- **ZigBee PRO**
  - Enhanced features optimize performance and RAM utilization under select scenarios
  - Feature enhancements based on identified limitations of ZigBee 2006 for specific network deployments

- **Interoperability**
  - PRO devices will operate as End Devices on a ZigBee ‘06 or ‘07 network, and vice versa
  - ZigBee ‘06 and ‘07 network seamlessly

**ZigBee 2007 Feature Set**

- **ZigBee**
  - CSKIP Tree Addressing
  - AODV Routing
  - Backup Tree Routing
  - Fragmentation
  - Frequency Agility
  - Basic Group Addressing
  - Security

- **ZigBee PRO**
  - Stochastic Addressing
  - AODV Routing
  - Many to One / Source Routing
  - Asymmetric Link Handling
  - Fragmentation
  - Frequency Agility
  - Basic Group Addressing
  - Limited Broadcast Addressing
  - Security
  - High Security

---

**ZigBee 2007 Features ...**
ZigBee 2007 Features

ZigBee 2007 - Automated Address Assignment

ZigBee: Tree Addressing Basics
- Uses CSKIP algorithm to dynamically distribute addresses
- Provides well defined topology for backup tree routing
- Supports mobile End Devices through Rejoin and ED Announce

ZigBee PRO: Stochastic Addressing Basics
- Randomly assigns addresses to joining devices
- Deals with potential address conflict through resolution protocol
- Adds extended address to network packets for resolution
- Supports mobile End Devices through Rejoin and ED Announce

ZigBee 2007 - Group Addressing

ZigBee: Basic Group Addressing
- Send broadcast out to entire network and filter by group ID at the application layer

ZigBee PRO: Basic + Limited Broadcast Group Addressing
- Basic group addressing is same as ZigBee
- Group member can send limited broadcast with restrictions on message propagation
- Limited broadcast restricts non-member re-broadcasts

Limited Broadcast Group Addressing Tradeoffs
- Network communication is reduced in very large networks where communication is mostly local
- Sleeping devices cannot participate in the group
- Message sender has to be a group member
- Ineffective in small networks or for cross-network communication
ZigBee 2007 - Fragmentation

**Basics**
- Optional in BOTH ZigBee and ZigBee PRO
- Used to disassemble and reassemble large application payloads
- Implemented at application layer as an add-on

**Tradeoffs**
- Allows application to send large packets without concern for size
- Adds code complexity and application header overhead

ZigBee 2007 - Frequency Agility

**Basics**
- Mandatory in BOTH ZigBee and ZigBee PRO
- Used to compensate for RF interference by monitoring network statistics and notifying ZDO when “problem” is detected
- ZDO of Network Manager responds accordingly

**Tradeoffs**
- Provides solution for interference prone environments
- Requires additional complexity maintaining network statistics
- Requires resynchronization when sleeping End Devices miss channel migration message
- Can be costly in terms of battery consumption
- Can be problematic if a Router misses the “change” message

ZDO = ZigBee Device Object (network and device management)
ZigBee 2007 Features

ZigBee 2007 PAN ID Conflict Resolution

**Basics**
- ZigBee PRO feature only
- Used to resolve the scenario when co-existing networks select the same PAN_ID on the same channel
- All devices monitor neighbor communication during joining to identify duplicate PAN
- Network Manager handles resolution, backup Manager can be designated

**Tradeoffs**
- When commanded to change PAN_ID, devices missing the message can become stranded
- Commissioning can be used to prevent this problem

PAN = Personal Area Network

ZigBee 2007 Security

**ZigBee: Basic Security**
- Secured network communication through AES-128 encryption

**ZigBee PRO: Basic + optional High Security**
- High Security creates a mechanism for establishing link keys between peer to peer connections
- High Security adds additional security when devices on the network may not be trusted (inside attack)

**High Security Tradeoffs**
- Requires the overhead of link key establishment
- Adds significant complexity consuming valuable code space
- SE and HA profiles chose NOT to use high security
- CBA will probably not use high security

AES-128 = 128-bit Advanced Encryption Standard
ZigBee 2007 Summary

**ZigBee** is useful …
- In most topologies; including Peer to Peer or Sensor Reporting

**ZigBee drawbacks include:**
- Limited address assignment capabilities in a mobile environment
- Route establishment takes time and expends energy during route establishment in many source to Concentrator scenario

ZigBee PRO Summary

**ZigBee PRO is useful in networks with:**
- Large deployments with a high ratio of mobile devices
- Many sensor nodes reporting to Concentrator
- High Security Requirements

**ZigBee PRO drawbacks include:**
- Additional features increase the code size and complexity
- Network will suffer reduced throughput due to communication overhead
- Heavy burden on the “Concentrator” device
- Required Network Manager becomes a Point of Failure
ZigBee Advantages and Architecture

ZigBee Sales Pitch

- Enable low cost, low power, reliable devices for monitoring and control
- Ensure that devices are efficient in their use of the available bandwidth
- Provide a platform and implementation for wirelessly networked devices
  - Make it easy to design and develop ZigBee devices
  - Reduce today's cost of building wireless solutions
- Enable "out-of-the-box" interoperable devices where desired by manufacturers

ZigBee Device Software Architecture

- Endpoint app 240
  - User app
- Endpoint app 1
  - User app
- Endpoint app 0
  - ZDO
- ZigBee Stack
- OSAL
- TI-MAC
- Physical HW (PHY)
- HAL/I/O

Mesh Network...
Mesh Networking and Components

Mesh Network Devices

- ZigBee Coordinator
  - Starts the Network
  - Routes packets
  - Manages security
  - Associates Routers and End Devices
  - Example: Heating Central

-zigBee Router
  - Routes packets
  - Associates Routers and End Devices
  - Example: Light

- ZigBee End Device
  - Sleeps most of the time
  - Can be battery powered
  - Does not route
  - Example: Light switch

- Devices are pre-programmed for their network function

Coordinator

- Starts a non-beaconed PAN
- Allows other devices to join it
- Buffers messages for sleeping End Devices
- Provides binding and address-table services
- Routes messages
- Dynamically repairs routing
- Can have I/O capability
- Manages security
- Radio always on
Router

- Does not own or start PAN (Scans to find a network to join)
- Allows other devices to join it after PAN has been started
- Routes messages
- Dynamically repairs routing
- Buffers messages for sleeping End Devices
- Support secure messaging
- Can have I/O capability
- Radio always on

End Device

- Does not:
  - route messages
  - own or start network
  - allow other devices to join it
- Scans to find a PAN to join
- Polls parent to get messages (can be disabled)
- Can be mobile
- Radio/CPU can sleep
PAN Formation

**PAN Formation - Coordinator**

- Form PAN
- Perform passive scan
- Energy detect each channel
- Scan confirmed
- Beacon request open channels
- Scan confirmed
- One channel is selected
- Process fully automated once started

Notify application via ZDO_STATE_CHANGE event

**PAN Discovery/Join - Router and End-Device**

- Discover PAN
- Energy detect each channel
- Scan confirmed
- Beacon request open channels
- Scan confirmed
- Every chip has a unique 64-bit IEEE address (used for joining ID)
- ZDO determines which PAN to join by PAN ID
- Process fully automated once started

List of discovered PANs are returned

Notify application via ZDO_STATE_CHANGE event
**Address Allocation**

- **CSKIP() Tree Address Allocation**

0x0000 Coordinator address 0xFFFF
- Coordinator child routers and .......... child end devices

0x0001 Router address – Depth 1 0x143D
- Router 0_0 child routers and .......... child end devices

0x0004 Router address – Depth 4 0x0017
- Router 3_0 child routers and .. child end devices

- **Home Automation Profile:**
  - MAX_CHILDREN = 20
  - MAX_ROUTERS = 6
  - MAX_DEPTH = 5

- Parents allocate addresses
- At Depth 5, a single address is allocated (if a router, no devices can join it)
- This is address assignment, not routing

-ZigBee Mesh Networking ...
Routing

ZigBee Mesh Routing

- Mesh network routing employs AODV (Ad Hoc On Demand Distance Vector Routing)
  - Ad Hoc (Network is unknown at start-up)
  - On Demand (Determines the route to the destination only when needed)
  - Distance Vector (Only the final destination and the next hop are stored at each node. Relies on a distributed protocol to handle routing)
- Self healing upon route failure
  - Reliable and robust. Failed router will reinitiate discovery and find an alternative path

Routing is a decentralized, cooperative process

- Routers (and Coordinator) forward unicast messages directly to the destination
- Other messages prompt the router to check its' routing table
  - If an entry exists, router forwards message to the next node
  - If none exists, route discovery takes place
- Route discovery searches all possible routes using request/response packets (RREQ/RREP)
  - "Route cost", a function of RSSI, to all neighbors is recorded
  - Routing algorithm selects and stores: the destination address, the next hop node address and the link status in the routing table
- Routes can be invalidated due to errors, expiration, mobile nodes or by the user, prompting a new route discovery
- Pro routing includes Many-to-One routing to optimize data concentrator traffic
Routing Table

- Every routing device contains a Routing Table
- The table stores information needed to route packets
- Routes can automatically expire if not used for ROUTE_EXPIRY_TIME seconds (f8wconfig.cfg)

Routing table entry:
- Destination address
- Next hop node
- Link status

- A Route Discovery table stores temporary information while route discovery is in process
- MAX_RREQ_ENTRIES (f8wconfig.cfg) sets the maximum number of simultaneous route discoveries that can be performed

Automatic Rerouting

1. Coordinator sends msgs to R3 via R1 (blue path), then R1 fails
2. Coordinator sends msgs to R3 via R2 (green path), then R2 fails
3. Coordinator sends msgs to R3 via R4 and R5 (red path)
Configuration

Configure the Channel in f8wConfig.cfg

```c
// * Default channel is channel 11 = 0x0A
// Channels are defined in the following:
// 0 = 868 MHz 0x00000001
// 1 - 10: 915 MHz 0x0000000F
// 11 - 26: 2.4 GHz 0x07FFFF00
//
// - Default Channels 868MHz 0x00000001
// - Default Channels 915MHz 0x0000000F
// - Default Channels 24GHz 0x07FFFF00
// - DEFAULT_CHANNEL=0x00000000 // 36 - 0x1A
// - DEFAULT_CHANNEL=0x00000000 // 21 - 0x11
// - DEFAULT_CHANNEL=0x00000000 // 24 - 0x18
// - DEFAULT_CHANNEL=0x00000000 // 23 - 0x17
// - DEFAULT_CHANNEL=0x00000000 // 22 - 0x16
// - DEFAULT_CHANNEL=0x00000000 // 21 - 0x15
// - DEFAULT_CHANNEL=0x00000000 // 20 - 0x14 // Default Channel
// - DEFAULT_CHANNEL=0x00000000 // 19 - 0x13
// - DEFAULT_CHANNEL=0x00000000 // 18 - 0x12
// - DEFAULT_CHANNEL=0x00000000 // 16 - 0x10
// - DEFAULT_CHANNEL=0x00000000 // 15 - 0x0c
// - DEFAULT_CHANNEL=0x00000000 // 14 - 0x0B
// - DEFAULT_CHANNEL=0x00000000 // 13 - 0x0a
// - DEFAULT_CHANNEL=0x00000000 // 11 - 0x08
```

Multiple channels may be selected

Configuring PAN ID in f8wConfig.cfg

```c
/* Define the default PAN ID. *
* Setting this to a value other than 0xFFFF causes *
* 206_CRYPTO to use this value as the PAN ID and *
* Routers and end devices to join PAN with this ID */

PAN_ID_CFG = 0x00000000
```

If `PAN_ID = 0xFFFF` and `device = Coordinator`:
Device uses IEEE address to choose a PAN_ID (last 2 bytes)

If `PAN_ID = 0xFFFF` and `device = Router` or `End Device`:
Device will join any available PAN

If `PAN_ID ≠ 0xFFFF` and `device = Coordinator`:
Device will use the set value for the PAN_ID

If `PAN_ID ≠ 0xFFFF` and `device = Router` or `End Device`:
Device will ONLY join a PAN that has this PAN_ID

Lab time ...
Lab8A – Starting a PAN

Description:

This lab utilizes the main sample application delivered with the ZigBee download: GenericApp. We’ll be modifying this example in the next few labs, so let’s get acquainted with getting the software running and what it looks like to start a PAN.

Lab 8A – Starting a PAN

- Open GenericApp project
- Change Channel and PAN ID parameters
- Configure and load
  - Coordinator
  - Router
  - End-device
- Run Packet Sniffer
- Start Coordinator, then Router then End Device and observe network traffic

<table>
<thead>
<tr>
<th>Group</th>
<th>Channel</th>
<th>PAN ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x0C (12)</td>
<td>0x0AAA</td>
</tr>
<tr>
<td>2</td>
<td>0x0D (13)</td>
<td>0x0BEE</td>
</tr>
<tr>
<td>3</td>
<td>0x0E (14)</td>
<td>0x0CEE</td>
</tr>
<tr>
<td>4</td>
<td>0x0F (15)</td>
<td>0x0DEE</td>
</tr>
<tr>
<td>5</td>
<td>0x10 (16)</td>
<td>0x0EEE</td>
</tr>
<tr>
<td>6</td>
<td>0x11 (17)</td>
<td>0x0EFF</td>
</tr>
<tr>
<td>7</td>
<td>0x12 (18)</td>
<td>0x0BAB</td>
</tr>
<tr>
<td>8</td>
<td>0x13 (19)</td>
<td>0x0ACE</td>
</tr>
</tbody>
</table>
Hardware list:

- 3 SmartRF05EB boards
- 3 CCMSP-EM430F2618 boards
- 3 CC2520EM boards
- 1 SmartRF04EB board
- 1 CC2430EM board
- 4 antennas
- MSP-FET430UIF and ribbon cable
- 5 USB A/B cables
- 2 AA batteries
- Post-it™ flags

Software list:

- IAR Embedded Workbench for MSP430 version 4.11D
- TI Packet Sniffer version 2.10.1

(You will find shortcuts for the above applications on the desktop)

- Z-Stack version 2.1.0 – 1.3.0
Procedure

1. Arrange and turn on the hardware

Three of the SmartRF05EB/CCMSP-EM430F2618/CC2520EM/antenna stacks should already be assembled and ready from the TI-MAC lab, like the photos below:

Label all three boards; one as a Coordinator, another as a Router and the last as an End Device.

Connect all three boards to the PCs USB port with a USB A/B cable (you may need a USB hub) and make sure that the Power Jumper on these two boards joins pins 2&3 (the rightmost pins). Remove any batteries from these boards and turn the boards off.

Your Sniffer board should already be assembled (SmartRF04EB/CC2430EM/antenna) like the photo below. Connect it to the PCs USB port with a USB A/B cable and turn the board off.

Connect the MSP-FET430UIF Emulator to the PCs USB port using the last USB A/B cable.

2. Start IAR Embedded Workbench

Start IAR Embedded Workbench and open the GenericApp.eww workspace from the following folder:

C:\Texas Instruments\ZStack-2.1.0-1.3.0\Projects\zstack\Samples\GenericApp\CC2520DB
3. **Configure the correct channel and PAN ID**

Open the file `f8wConfig.cfg` in the **Tools** group and configure your workgroups assigned **channel** and **PAN ID**. A common mistake in this step is to leave more than one channel enabled. Although it’s okay to do this, the ZigBee stack software will select the best channel (among the ones you’ve enabled) to operate on. That will make your transmissions hard to find with the Sniffer and you may transmit on another workgroups assigned channel.

4. **Turn Automatic Polling Off**

If we leave automatic polling on, the **End Device** will generate periodic data requests to the **Coordinator** and make it difficult for us to see the transmissions we actually want to watch. Turn automatic polling off by assuring that `-DPOLL_RATE=0` in `f8wConfig.cfg`. This option is located near the bottom of the file.

5. **Build and Download**

**Connect** the Emulator to the **debug port** on the **Coordinator**. Turn the **Coordinator on**. If you are prompted to complete the Hardware Wizard, do so and install the software automatically. In IAR Embedded Workbench, select the **Coordinator** configuration from the pull-down menu as shown below:

```
Workspace
| Coordinator
| Coordinator
| Router
| EndDevice
| CoordinatorPro
| Router-Pro
| EndDevice-Pro
```

**Build** and **download** the project to the Coordinator. When then download is complete, click the **Stop Debugging** button and turn the board **off**.

**Repeat** the procedure for both the **Router** and **End Device**. When you are done, minimize IAR Embedded Workbench.
6. Run the Packet Sniffer

Turn on your sniffer hardware, and then run the Packet Sniffer. Select the protocol/chip type as IEEE 802.15.4/ZigBee (CC2430) and click Start. Make sure to select your assigned channel and click the Capture button.

7. Start the network

Note: If, during any of the following steps, the Green LED is blinking after powering up a device it means the device does not have a pre-assigned IEEE address. Press straight down on the joystick (Switch 5) to assign a random IEEE address dynamically. Without an IEEE address, the network cannot build.

Turn on the Coordinator and watch the LCD. More than likely you’ll see the message “Energy Level Scan Failed”. Remember that part of the network start-up process involves passive energy detection on the available channels. There is a pre-set energy level called ZDNWKMG1R_ACCEPTABLE_ENERGY_LEVEL, above which the code determines a failure. For this lab, though, we have set up the network to operate on a single channel. There’s no choice but to start the network on this channel. On the Sniffer, note the beacon request during the active scan. The Coordinator should assign itself address 0x0000.

Turn on the Router. Note the Association dialog and the short address assigned by the Coordinator. Since this is the first Router associated to the Coordinator, it should receive address 0x0001. We are not using ZigBee PRO in these labs, so the CSKIP addressing method is running, not stochastic addressing.

Turn on the End Device. It should associate with the Coordinator and, as the first End Device associated with it, should receive address 0x796F.

Note the status displayed on the LCD display of each board and that LED D3 is lit, indicating that the Z-Stack has started and initialized.
Send Data

8. Send some data

The code within `GenericApp.c` is now waiting for us to bind the applications (running on each of the boards) together so that some data can be sent. Two different types of binding are implemented:

**SW2** is Assisted Binding  
**SW4** is Auto Match

We’ll cover binding in detail later in the workshop. For now, the only thing that matters is that we bind the applications together so that some periodic data can be sent.

On the **End Device**, move the joystick **left** (SW4). **LED4** should light on the board and both the **Coordinator** and **Router** will display **Hello World** on their LCD displays. The application on the **End Device** has bound to both the **Router** and **Coordinator** based applications and will send **Hello World** across the network every 5 seconds. The auto-match binding type can only store a single address though … since the Router response was probably received after the Coordinator response, you’ll only see messages bound for the Router. Note the MAC-level acknowledgements too.

9. You’re finished

Shut down the Coordinator, Router and End Device boards. Shut down the sniffer software too. We’ll use this same hardware in the next lab.

You’re done.
Profiles

ZigBee Alliance defines Profiles as a method to ensure application level interoperability:
- Defines device types in the profile
- Agrees on message format, content coding, and interpretation of clusters
- Specifies network stack profile
- A profile ID must be unique and is issued by the alliance
- Proprietary profiles can be defined by the customer
  - limits application interoperability to devices that share this profile.
  - Does not exclude network level interoperability

Status of Alliance Profiles (10/2008):
- **Home Automation (HA)**
  - Released for ZigBee and ZigBee PRO
- **Smart Energy (SE)**
  - Formerly Advanced Metering Infrastructure (AMI)
  - Released for ZigBee PRO
- **Commercial Building Automation (CBA)**
  - Under development
  - Adopted as standard for wireless BACnet
- **Personal Health and Hospital Care (PHHC)**
  - Under development

Clusters...

Switch  Blind
Light  Sensor
Thermostat Pump
Clusters

Values sent or received over the network are called **Clusters** (data types).

The cluster library contains a complete list of these data types. For HA, see `zcl_ha.h`.

**Endpoint** applications send messages to other Endpoint applications using clusters.
Endpoints

- An **Endpoint** is the address of your application
  - 0 is the ZDO application address
  - Up to 240 additional applications can be supported on a device
  - Each endpoint is identified by its own descriptor structures

- This device contains two endpoints (in addition to the ZDO)
- Each endpoint contains a switch application

Descriptors...
Descriptors

- Populated in your application initialization code
- Must be registered with the Application Framework
- See AF.h for definitions of these structures
- Simple and Endpoint descriptors provide information about the endpoint applications
- Power and Node descriptors provide information about the device

Simple Descriptor

```c
typedef struct
{
    byte Endpoint;
    byte AppProfId;
    byte AppDeviceId;
    int AppDevVer:4;
    int Reserved:4;
    byte AppNumInClusters;
    cId_t *pAppInClusterList;
    byte AppNumOutClusters;
    cId_t *pAppOutClusterList;
} SimpleDescriptionFormat_t;
```

- One per Endpoint
  - Endpoint number (1 – 240)
  - Profile ID from Alliance
  - Device ID from Alliance
  - Version
  - Number of input clusters
  - Pointer to input cluster list
  - Number of output clusters
  - Pointer to output cluster list
**Endpoint descriptor**

```c
typedef struct {
    byte endPoint;
    byte *task_id;
    SimpleDescriptionFormat_t *simpleDesc;
    afNetworkLatencyReq_t latencyReq;
} endPointDesc_t;
```

- **One per Endpoint**
  - Endpoint number (1 – 240)
  - OSAL application task ID
  - Pointer to simple descriptor
  - 0

Each Endpoint must be registered with the application framework (AF):

```c
afRegister( endPointDesc_t *epDesc )
```

**Node Power Descriptor**

```c
typedef struct {
    unsigned int PowerMode:4;
    unsigned int AvailablePowerSources:4;
    unsigned int CurrentPowerSource:4;
    unsigned int CurrentPowerSourceLevel:4;
} NodePowerDescriptorFormat_t;
```

- **One per device**
- **PowerMode**
  - Receiver always on, periodically on or externally activated
- **PowerSources**
  - Constant (Mains)
  - Rechargeable batteries
  - Disposable batteries
- **CurrentLevel**
  - Critical, 33%, 66%, 100%
Node Descriptor

typedef struct
{
    uint8 LogicalType:3;
    uint8 ComplexDescAvail:1;
    uint8 UserDescAvail:1;
    uint8 Reserved:3;
    uint8 APSFlags:3;
    uint8 FrequencyBand:5;
    byte CapabilityFlags;
    uint8 ManufacturerCode[2];
    uint8 MaxBufferSize;
    uint8 MaxInTransferSize[2];
    uint16 ServerMask;
    uint8 MaxOutTransferSize[2];
    uint8 DescriptorCapability;
} NodeDescriptorFormat_t;

- One per **device**
- Coordinator, Router or End Device
- Band: 868MHz, 902MHz or 2.4GHz
- Max single packet size
- Network Manager, Primary/Backup Trust center, binding table, or discovery table
Architecture

ZigBee Device Architecture

- Endpoint app 240
  - User app

- Endpoint app 1
  - User app

- Endpoint app 0
  - ZDO

Application Framework

ZigBee Stack

MAC

Physical HW

HAL/I/O

OSAL
The AF provides applications with structures and functions to:
- Manage endpoints
- Send and receive data

**Address Structure**

```c
typedef struct{
    union{
        uint16 shortAddr;
        ZLongAddr_t extAddr;
    } addr;
    afAddrMode_t addrMode;
    byte endPoint;
} afAddrType_t;
```

- Unicast to 16-bit short or
- 64-bit extended destination address
- Addressing mode
- Endpoint number

**Mode Parameter**

- Addr16Bit: Unicast
- AddrNotPresent: Indirect – destination address found in binding table
- AddrBroadcast: Broadcast to all devices, non-sleeping devices or routers/Coordinator only
- AddrGroup: Devices can assign themselves to groups addressable here
Send Data Function

```c
afStatus_t AF_DataRequest(
    afAddrType_t *dstAddr,
    endPointDesc_t *srcEP,
    uint16 cID,
    uint16 len,
    uint8 *buf,
    uint8 *transID,
    uint8 options,
    uint8 radius );
```

- Pointer to address structure
- Endpoint descriptor of sending endpoint
- Cluster ID for this message
- Message length in bytes
- Pointer to message data
- Transaction sequence number
- Transmit options (i.e. ack request)
- Max propagation radius in hops

Return parameters defined in ZComDef.h

- afStatus_SUCCESS
- afStatus_FAILED
- afStatus_MEM_FAIL
- afStatus_INVALID_PARAMETER

Receive data interface ...

Receiving Data

1. After you register the Endpoint with the AF:
   ```c
   afRegister( &GenericApp_epDesc );
   ```
   you can receive incoming OTA message callbacks at that Endpoint

2. Your Task event handler is triggered by the SYS_EVENT_MSG event:
   ```c
   case AF_INCOMING_MSG_CMD:
      GenericApp_MessageMSGCB( MSGpkt );
      break;
   ```

3. Now your message callback routine can process and act on the message:
   ```c
   void GenericApp_MessageMSGCB( afIncomingMSGPacket_t* pkt )
   { switch ( pkt->clusterid )
     { case GENERICAPP_CLUSTERID:
       #if defined( LCD_SUPPORTED )
       HalLcdWriteScreen( (char*)pkt->cmd.Data, "rcvd" );
       #endif
       break;
     }
   }
   ```

Received Data Structures

```c
typedef struct
{ byte TransSeqNumber;
  uint16 DataLength;
  byte *Data;
} afMSGCommandFormat_t;
```

Received Data Structures

```c
typedef struct
{ osal_event_hdr_t hdr;
  uint16 groupid;
  uint16 clusterid;
  afAddrType_t srcAddr;
  byte endPoint;
  byte wasBroadcast;
  byte LinkQuality;
  byte SecurityUse;
  uint32 timestamp;
  afMSGCommandFormat_t cmd;
  afIncomingMSGPacket_t;
} afIncomingMSGPacket_t;
```

Visualizing Acknowledgments ...
Visualizing Acknowledgements

- MAC-level acknowledgements & retries are default and automatic regardless of service used. Destination app is unaware and uninvolved.
- Specifying ACK generates an ACK by the destination AF task.
- If you are using application level acknowledgments, `AF_DATA_CONFIRM` will notify you of the complete transmissions' success/failure.
- If you are not using application level acknowledgments, `AF_DATA_CONFIRM` will notify the app of the first hops' success/failure.
Third Party Tools

**Daintree Sensor Network Analyzer**

ZigBee can be a bit confusing when you’re getting started. The SNA shows the network in an easy-to-understand format with devices and interactions shown graphically, and ability to drill down for additional detail to provide a more complete picture of what is going on.

![Daintree Sensor Network Analyzer](image)

**Keeping you up to date**

The SNA stays up to date, not only with the latest ZigBee standards, but also with the latest functionality offered by TI.

For example, recent releases of the SNA support commissioning, binding, and TI-specific features including over-the-air firmware upgrades and locationing. The SNA provides full support for ZigBee 2006, 2007 and PRO.
Complete and Compliant

Help to ensure your products pass ZigBee certification at the first attempt.

The Daintree SNA is used by the ZigBee test houses as the primary platform-independent means of visualizing, recording and verifying conformance to ZigBee specifications and test plans.

Your TI ZigBee development kit, including the SNA, provides everything you need for development, testing and field trials of ZigBee devices.

The Basic edition of the SNA comes standard with the TI kits, together with 30-day trial versions of the enhanced Standard and Professional editions. Find out more at www.daintree.net

Atalum Manage & Commission Suite

Network Summary at a Glance

Synch-up

Easy to use Windows tool for Commissioning Made 1-2-3

Graphical Representation of Multiple Networks
Maximize a ZigBee Infrastructure

- Atalum enables companies to maximize a ZigBee infrastructure by minimizing maintenance & management costs.
- A ZigBee infrastructure provides a standard wireless means to control AND monitor ANY thing in ANY place.
- To realize the potential of a ZigBee infrastructure companies should have the means to remotely access & automate what’s going on in the network.
- Plus, commissioning & on-going management must be at least as economical as existing systems.
- Companies should also be able to integrate the information from ZigBee networks with existing systems and seamlessly interface the data across applications, locations & users.
- Also, installers require a way to quickly commission ZigBee devices without any special expertise.
- Regardless of the phase in the life-cycle Atalum’s GREENsuite provides the necessary functionality to deploy, manage and maintain a ZigBee network in a cost-effective and easy way.

Fully Loaded Application Suite

- Atalum’s GREENsuite provides the tools required by installers, SIs & OEMs.
- Atalum offers a Web network management application along with a Windows commissioning tool. Information can be synched between the two applications. APIs for integration with legacy or other systems can also be used.
- Atalum’s intuitive GUIs make it easy to deploy and maintain multiple hardware platforms.

- **GREENsuite** allows companies to:
  - Set-up Networks & Configure Sensors
  - Register Devices
  - Schedule Events
  - Define Triggers & Alarms
  - View Network Diagnostics & Performance
  - Obtain Reports & Statistics (User Defined Queries or Default)
  - Manage Users (Access & Privileges)
  - Obtain System Logs (Summary or Detail)
  - Export Information (Multiple Formats)
  - Integrate with Legacy, Billing or Other Systems
Low Power RF Developer Network

- Modules
- Engineering Services
- Development Tools

Go to www.ti.com/zigbee and click on:

Low Power RF Developer Network - the latest news of the developer program...
Writing to the LCD

- Set of 6 APIs within the HAL, e.g. ...

Write a string to the LCD
extern void HallcdWriteString ( char *str, uint8 option);

Write a value to the LCD
extern void HallcdWriteValue ( uint32 value, const uint8 radix, uint8 option);
Writing to the LCD

*** blank left intentionally page this ***
Lab 8B – Code Walkthrough

Description:

From an application perspective of the current state of GenericApp, all three nodes are programmed as the same device, in the same profile, residing on the same endpoint and sending the same data type, or cluster. We’re simply using conditional compilation to segment the parts we want to use on each different device.
Hardware list:

- 3 SmartRF05EB boards
- 3 CCMSP-EM430F2618 boards
- 3 CC2520EM boards
- 1 SmartRF04EB board
- 1 CC2430EM board
- 4 antennas
- MSP-FET430UIF and ribbon cable
- 5 USB A/B cables
- 2 AA batteries
- Post-it™ flags

Software list:

- IAR Embedded Workbench for MSP430 version 4.11D
- TI Packet Sniffer version 2.10.1
- Daintree Sensor Network Analyzer version 2.3.0.8

(You will find shortcuts for the above applications on the desktop)

- Z-Stack version 2.1.0 – 1.3.0
Procedure

1. Start IAR Embedded Workbench and load Project

Start IAR Embedded Workbench from the desktop shortcut. When the Startup dialog appears, select Open existing workspace and navigate to: C:\Texas Instruments\ZStack-2.1.0-1.3.0\Projects\zstack\Samples\Lab8B\CC2520DB. Select GenericApp.eww and click Open.

The Lab8B project is simply a copy of the GenericApp project we used in the previous exercise.

2. Set your channel, PAN ID and turn off polling

Refer back to LAB8A steps 3 and 4. Set the channel and PAN ID to those assigned to your workgroup. Assure that End Device polling is off.

3. Open GenericApp.h

In the App workspace group, open GenericApp.h and note the definitions of GENERICAPP_ENDPOINT, _PROFID, _DEVICEID and _CLUSTERID. These definitions place the application at endpoint 10, using profile ID 0x0F04, device ID 0x0001 and cluster ID 1. This is only an example profile, etc, but remember that those numbers will have to match for applications to communicate with each other. In a real application, the profile ID would have been obtained from the ZigBee Alliance.

4. Open GenericApp.c

Open GenericApp.c and navigate to the GLOBAL VARIABLES area, where GenericApp_ClusterList is established and the Simple Descriptor structure is defined. Unless you have a short-term memory issue, this structure should look pretty familiar from the presentation. It describes the Endpoint being used on all 3 network devices.

5. Warning: Code Explanation Ahead

GenericApp.c is broken into two main functions:

- GenericApp_Init()
- GenericApp_ProcessEvent()

And four local functions:

- GenericApp_ProcessZDOMsgs()
- GenericApp_HandleKeys()
- GenericApp_MessageMSGCB()
- GenericApp_SendTheMessage()
GenericApp_Init is run by the OSAL when GenericApp is initialized in OSAL_GenericApp.c. It populates the Endpoint Descriptor structure, registers the endpoint description with the Application Framework, registers for keypress callback, writes a message on the LCD and then registers for two ZDO binding-type callbacks; an end device bind response and a match descriptor response. Note the if defined statement above the LCD write … this is how various features are included and excluded based on compiler preprocessor defines.

GenericApp_ProcessEvent is the run-time part of task GenericApp. It receives a callback whenever a system event (manifested by SYS_EVENT_MSG) or an application event (manifested by GENERICAPP_SEND_MSG_EVT) has occurred. A message parser runs a case statement based on what type of message was received.

If a SYS_EVENT_MSG occurred and the message is a:

- ZDO callback – GenericApp_ProcessZDOMsgs is run (more on the ZDO later in the class)
- Key change – GenericApp_HandleKeys is run
- AF_DATA_CONFIRM_CMD – actions are taken for a transmitted message acknowledgement
- AF_INCOMING_MSG_CMD – GenericApp_MessageMSGCB is run to process the incoming message
- ZDO_STATE_CHANGE – Binding has been successful. We can start the outgoing message timer

If a GENERICAPP_SEND_MSG_EVT occurred (because of the OSAL timer expiring), a new message is sent and the OSAL timer is restarted.

GenericApp_ProcessZDOMsgs():
Lights or blinks LED4 based on success/failure of the end device bind request (more on binding later). Binding information is automatically stored.
Lights LED4 based on success of a match descriptor request. Stores binding information.

GenericApp_HandleKeys():
If switch 2 has been pressed, send an end device bind request (assisted binding)
If switch 4 has been pressed, send a match descriptor request (automatic binding)

GenericApp_MessageMSGCB():
 Parses the incoming message data and displays it on the LCD

GenericApp_SendTheMessage():
Constructs and sends (via an AF_DataRequest) the “Hello World” message over the air

Build and download the correct project configurations to the three network boards. When you’re finished, **power them down** and minimize IAR Embedded Workbench.

Power up the SmartRF04EB/CC2430EM stack, **start** the Packet Sniffer and select the **IEEE 802.15.4/ZigBee (CC2430)** protocol. Set your assigned **channel** and **start** the Packet Sniffer capturing.

In order, power up the; **Coordinator**, **Router** and **End Device**. Observe the association process on the Packet Sniffer.

Following **step 8** from **Lab8A**, press **SW4** (joystick left) on the **End Device**. **LED4** will light on the board, indicating successful binding with the applications on the Coordinator and Router. The application on the End Device will now send the “Hello World” message to the other boards every 5 seconds. Remember that we should only see traffic to the Router because we used auto-match binding.

Using the Packet Sniffer, observe and verify the data traffic.

Close the Packet Sniffer and shut down all four boards. Maximize IAR Embedded Workbench.

7. Modifying the LCD display

In addition to seeing the incoming message displayed on the screen, it would be nice to know how many messages have arrived.

In **GenericApp.c**, look in the **GenericApp_MessageMSGCB** function. Beneath the **HalLcdWriteScreen()** statement, add the following code. For those of you who don’t like to type, I’ve included Lab8B_Code.txt that you can cut/paste from.

```c
    count++;  
    HalLcdWriteValue ( count, 10, HAL_LCD_LINE_3);
```

8. Add local variable

In the **Local Variables** area at the top of **GenericApp.c**, add the following variable definition:

```c
    uint16 count=0;
```

9. Build, download and test

Follow the procedure in **step 6** above. Verify that the LCD updates properly. Note that the Coordinator completes the binding response and the End Device application sends a “Hello World” message to the Coordinator. Quickly after that, the Router is able to send its’ response to the End Device (through the Coordinator). It then becomes the bound device since it was last to respond. On the LCD’s, you should see a single message appear on the Coordinator and then the message count should increment on the Router. This is expected behavior.
Application Level Acknowledgements

Up to this point from an application perspective, a device sends data over the network to another device to which it is bound (more on binding later). As far as the application knows, the device it is communicating with could be 5 feet away, or on the other side of the building where communication requires several hops to get there. The MAC level acknowledgements that automatically occur would only inform us of the successful reception of the first hop. To obtain application level confirmation that a packet was received, ZigBee provides optional application level acknowledgements.

10. Add application acknowledgements

In GenericApp.c, find the GenericApp_SendTheMessage() function. Note that the options parameter in AF_DataRequest is the second to last parameter, and AF.h defines the bitmap for the options parameter. Make sure that when adding the AF_ACK_REQUEST option you do not eliminate the already existing AF_DISCV_ROUTE option.

```c
if ( AF_DataRequest( &GenericApp_DstAddr, &GenericApp_epDesc, 
    GENERICAPP_CLUSTERSID, 
    (byte)osal_strlen( theMessageData ) + 1, 
    (byte *)&theMessageData, 
    &GenericApp_TransID, AF_ACK_REQUEST | AF_DISCV_ROUTE, 
    AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
```

11. Build, Download and Test

Follow the procedure in step 6, above, and observe the difference in traffic on the Packet Sniffer.
12. Daintree Sensor Network Analyzer

The Daintree SNA includes many more features and advanced tools than the TI Packet Sniffer. Let’s fire it up now.

Shut down the three network boards and the Packet Sniffer. Start the Daintree SNA from the desktop shortcut. Select the SmartRF04EB as the Source and select your workgroup’s assigned channel.

Click the Start Capture button, and then click Window → Tile Default Layout from the menu bar.

Observe the SNA as you power up the Coordinator, Router and End Device in order. Press SW4 (joystick left) to bind the application.

Change the view in the Visual Device Tree window to Application and watch the data (green) and acknowledgement (red) transmissions occur. Click on either line in the display to view additional information.

Double-click on a packet in the Packet List or Packet Timeline window to view additional details in the Packet Decode window.

13. You’re finished

Shut down the Daintree SNA (click Yes, then No to the nag boxes) and IAR Embedded Workbench. Power off all four boards.

You’re done.
Digging Further into the Stack

A Simple Network

- The light switch device has 2 endpoints
- The lamp device has 4 endpoints
- Binding can be one to one, one to many or many to many

Binding ...
Binding

Applications bind with other applications.
Simple and Endpoint descriptors are used to determine who can bind.
IN clusters and OUT clusters must have the same profile ID and be compatible.
Documented in Z-Stack API Guide and in Z-Stack Developer’s Guide.

There are 4 types of binding:
- Automatic
- Assisted
- Centralized
- Application

Defined in RAM, but can be saved in Flash if the NV_RESTORE compiler option is used.
Stored on source node (REFLECTOR compiler option required).
Enteries map messages to their intended destination.
Each entry in the binding table contains the following:

```c
typedef struct
{
    uint16 srcIdx;  // Source index
    uint8 srcEP;    // Source endpoint
    uint8 dstGroupMode; // Specifies normal or group addressing
    uint16 dstIdx;  // Destination index or group address
    uint8 dstEP;    // Destination endpoint
    uint8 numClusterIds; // Number of cluster IDs in the clusterIdList below
    uint16 clusterIdList[MAX_BINDING_CLUSTER_IDS];
} BindingEntry_t;
```

Automatic Binding...
Automatic Binding

- Sending device broadcasts a “personal ad” on the network with:
  - Address, Profile ID, Cluster Lists
  - Match Description Request - ZDP_MatchDescReq()
- Compatible devices respond
- Response handled and validated by the ZDO
- Sender application stores binding record in binding table
- Sometimes called “Service discovery”, “AutoFind” or “AutoMatch”

Assisted Binding

- External device initiated binding (“external” meaning not a participant of the resultant binding)
- External device application calls ZDP_BindReq() with 2 applications (addresses and endpoints) and the cluster ID to bind. The first parameter is where the binding record will be stored
- Make sure you have REFLECTOR compile flag enabled

Centralized Binding ...
Centralized Binding

- Application initiates ZDP_EndDeviceBindReq() (i.e. via button press) with 2 applications (addresses and endpoints) and the cluster ID to bind. The first parameter is where the binding record will be stored (sender).
- Coordinator receives and holds the request until a request from another node is received (16 seconds is the default)
  - Profile IDs must match, clusters must be compatible
- Make the REFLECTOR compile flag enabled
- Known as “source binding” in ZigBee 2007

Application Binding

- Applications can manage the binding table itself, adding and removing entries

- bindAddEntry() – Add entry to binding table
- bindRemoveEntry() – Remove entry from binding table
- bindRemoveClusterIdFromList() – Remove a cluster ID from an existing binding table entry
- bindAddClusterIdToList() – Add a cluster ID to an existing binding table entry
- bindRemoveDev() – Remove all entries with an address reference
- bindRemoveSrcDev() – Remove all entries with a referenced source address
- bindUpdateAddr() – Update entries to another address
- bindFindExisting() – Find a binding table entry
- bindIsClusterIdInList() – Check for an existing cluster ID in a table entry
- bindNumBoundTo() – Number of entries with the same address (source or destination)
- bindNumOfEntries() – Number of table entries
- bindCapacity() – Maximum entries allowed
- bindWriteNV() – Update table in NV

Which binding method should you use?
## Which Binding Method To Use?

<table>
<thead>
<tr>
<th>Method</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| **Automatic** | + no user interaction required  
+ no tool cost  
- development time knowledge  
- non-configurable |                                                                      |
| **Assisted**  | + install-time decisions (site-specific knowledge)  
+ analysis, maintenance, modification, visualization  
  can be under installers control  
- cost of tool |                                                                      |
| **Centralized**  | + allows user to decide  
+ cost of tool minimal  
- few, if any, configurable parameters  
- requires a user interface on each device |                                                                      |
| **App**  | + maximum flexibility  
- you must write all the code |                                                                      |
Z-Tool

- A PC application for sending and receiving commands to/from programs on the target board using the RS-232 port (transceiver required)
- Requires MT_TASK Compiler option

Sending Commands from Z-Tool

Receiving commands in app ...
Receiving Commands In Application

```c
byte MTProcessAppMsg( byte *pData, byte len )
{
    byte ret = ZFailure;
    byte endpoint;
    endPointDesc_t *epDesc;
    mtSysAppMsg_t *msg;

    // Get the endpoint and skip past it.
    endpoint = *pData++;
    len--;
    // Look up the endpoint
    epDesc = afFindEndPointDesc( endpoint );
    if ( epDesc )
    {
        // Build and send the message to the APP
        msg = (mtSysAppMsg_t *)osal_msg_allocate( sizeof( mtSysAppMsg_t ) + len );
        if ( msg )
        {
            msg->hdr.event = MT_SYS_APP_MSG;
            msg->endpoint = endpoint;
            msg->appDataLen = len;
            msg->appData = (uint8*)(msg+1);
            osal_memcpy( msg->appData, pData, len );
            osal_msg_send( * (epDesc->task_id), (uint8 *)msg );
            ret = ZSuccess;
        }
    }
    return ret;
}
```

- Message handler takes the incoming MT message and constructs an OSAL message for the application
- Handler code is automatically included when enabled

Sending Messages From Application

- Use `debug_str` to send a null-terminated string from within app
- Include Debugtool.h to your project

```c
#if defined( BN_DISPLAY_TEST ) && defined( MT_TASK )
    debug_str( "Not Sorted" );
#endif
```

- Values can be converted for display using `_ltoa()` [defined in osal.c]
  - Converts a long value to ascii
  - Declare: unsigned char buf[8];

```c
#if defined( BN_DISPLAY_TEST ) && defined( MT_TASK )
    _ltoa((unsigned long) ValueToConvert, &buf[0], 10);
    debug_str( buf );
#endif
```
Receiving Messages in Z-Tool

- Z-Tool automatically formats and displays incoming and outgoing messages
- Z-stack is instrumented to automatically provide system status at startup (when enabled)

Z-Tool Scripting

- Provides automated testing support
- JScript based script compiled by Z-Tool
- See Z-Tool Help file

```javascript
function MessageHandler(portName:String, id:MESSAGE_ID, msg:Object)
{
    switch(currentState)
    {
        case TEST_STATE.STATE1:
        {
            // set state to STATE2
            currentState = TEST_STATE.STATE2;

            // send SYS_LED_CONTROL command to Device2
            var req : SYS_LED_CONTROL = new SYS_LED_CONTROL();
            req.LedID = SYS_LED_CONTROL.LED_NUM.LED_1;
            req.Mode = SYS_LED_CONTROL.LED_MODE.BLINK;
            ZEngine.Send("Device2", req);
        }
    }
}
```
*** just where are the Elysian fields? ***
Lab 8C – Clusters, Binding and Z-Tool

- Add code to change reporting period
- Add a new control cluster
- Use centralized binding
- Add Z-Tool instrumentation

<table>
<thead>
<tr>
<th>Group</th>
<th>Channel</th>
<th>PAN ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x0C (12)</td>
<td>0x0AAA</td>
</tr>
<tr>
<td>2</td>
<td>0x0D (13)</td>
<td>0x0BEE</td>
</tr>
<tr>
<td>3</td>
<td>0x0E (14)</td>
<td>0x0CEE</td>
</tr>
<tr>
<td>4</td>
<td>0x0F (15)</td>
<td>0x0DEE</td>
</tr>
<tr>
<td>5</td>
<td>0x10 (16)</td>
<td>0x0EEE</td>
</tr>
<tr>
<td>6</td>
<td>0x11 (17)</td>
<td>0x0EFF</td>
</tr>
<tr>
<td>7</td>
<td>0x12 (18)</td>
<td>0x0BAB</td>
</tr>
<tr>
<td>8</td>
<td>0x13 (19)</td>
<td>0x0ACE</td>
</tr>
</tbody>
</table>

More presentation material ...
Hardware list:

- 3 SmartRF05EB boards
- 3 CCMSP-EM430F2618 boards
- 3 CC2520EM boards
- 1 SmartRF04EB board
- 1 CC2430EM board
- 4 antennas
- MSP-FET430UIF and ribbon cable
- 5 USB A/B cables
- 2 AA batteries
- Post-it™ flags

Software list:

- IAR Embedded Workbench for MSP430 version 4.11D
- TI Packet Sniffer version 2.10.1
- Daintree Sensor Network Analyzer version 2.3.0.8
- Z-Tool version 2.0

(You will find shortcuts for the above applications on the desktop)

- Z-Stack version 2.1.0 – 1.3.0
Procedure

The mechanisms for ZigBee device discovery and binding are managed outside of the user application in two places; the Application Framework (AF) and the ZDO. To further understand how devices discover and communicate with one another, we need to understand how the AF and ZDO participate together in the binding process.

Application Framework

The first component of interest, the Application Framework (AF), handles the de-multiplexing of incoming messages to the appropriate application. For this reason the user application must register with the AF during initialization. All of the information needed by the AF for this process is bundled into a Descriptor called the Endpoint Descriptor.

ZDO

The ZDO is a mandatory application component that handles management responsibilities such as how to create, discover, and/or join a network, how to perform binding, manage security, etc… The ZDO resides at Endpoint 0, and is implemented across several files that can be found in the ZDO group of your workspace.

The ZDO has been defined to minimize or even eliminate the customers’ need to implement or modify network management code. As you continue to build your ZigBee product(s), you may find that there are some features of the ZDO that need to change. When this happens we suggest you first look through the provided documentation, check the developer site for answers to FAQs, and finally contact support (support@ti.com), explain your requirements and ask for recommendations or assistance.

Lab

In this lab, the primary use of the ZDO (aside from the processes of network formation, discovery and joining which takes place automatically) is for binding. One method of binding, automatic, is initiated by calling the ZDP_EndDeviceBindReq() function and passing the Endpoint Descriptor as input. Once initiated, the autofind procedure utilizes ZigBee defined discovery mechanisms to match clusters between cooperating application components on different nodes. In the case of the automated binding procedure, the response to a binding request is passed back to the application in the form of a ZDO_STATE_CHANGE event, which provides the application with the information necessary to communicate.
Changing the reporting period

The current code sends data periodically (for both the Router and End Device) at a period set at compile time. In this section we are going to add code to allow the End Device (acting as a remote) to dynamically configure the reporting interval of the Router (acting as a sensor). To do this we are going to need to introduce a new message type, or Cluster. In addition to the first mechanism for binding the data clusters (Router -> Coordinator and End Device -> Coordinator), we are going to add a second binding mechanism that uses Centralized binding to bind the configuration cluster so that End Device can control the Router (End Device -> Router).

**Note:** We have implemented our code so that the Coordinator, Router, and End Device all run using the same application. This is done so that we don’t have to deal with managing two or three separate workspaces simultaneously, constantly opening and closing them for modifying and downloading code to different devices. We will continue to do so, but please take care in noting when you are adding behavior on the Coordinator (which is acting as the information sink and point of control) and the Router and End Device (which are acting as data sources). To differentiate code we will use the compile option `COORDINATOR` for further code modifications. We also use the compile options `ROUTER` and `ENDDEVICE`.

1. **Start IAR Embedded workbench and load the project**

   Start IAR Embedded Workbench and open `GenericApp.eww` from `C:\Texas Instruments\ZStack-2.1.0-1.3.0\Projects\zstack\Samples\Lab8C\CC2520DB`. This folder contains a copy of the solution from the previous lab.

   Open `GenericApp.h` for editing.

2. **Change the sending period from a #define**

   Change the `#define` for the `GENERICAPP_SEND_MSG_TIMEOUT` in `GenericApp.h` to be `SEND_MSG_TIMEOUT_DEFAULT`:

   ```
   #define SEND_MSG_TIMEOUT_DEFAULT   5000
   ```

   Open `GenericApp.c` for editing, and add a local variable:

   ```
   uint16 SendMsgTimeout;
   ```

   At the beginning of function `GenericApp_Init()`, initialize the variable to the default value as shown below. Check out `Lab8C_Code.txt` if you want to cut/paste.

   ```
   SendMsgTimeout = SEND_MSG_TIMEOUT_DEFAULT;
   ```

   Update the timer initialization calls in the `GenericApp_ProcessEvent()` function (both of them), to use the locally defined variable instead of the `#define`.

   ```
   osal_start_timerEx( GenericApp_TaskID, GENERICAPP_SEND_MSG_EVT, SendMsgTimeout );
   ```
3. **Build, load and observe automated binding**

As before, compile and download the correct configuration to each labeled board. If you have any problems with compiling the code, please verify that all variable names are correct.

Start the Packet Sniffer first, and then turn on the Coordinator, followed by the Router. Wait until the yellow LED is on for the Router (indicating network association) and then finally turn on the End Device. The yellow LED of the End Device should also turn on. Once devices are successfully bound you can stop the Packet Sniffer and observe the traffic where binding took place.

| Note: Remember that the boards require an IEEE address for networking to function. |

The Auto Find call initiates the transmission of a **Match Descriptor Request** by the device performing binding. This Match Descriptor Request is followed with a **Match Descriptor Response** unicast back to the Requester by all devices that have a binding match based on the provided Cluster information.
Adding a Control Cluster

Changing the reporting period

Binding occurs between endpoints, so to simplify the logic of binding for report data and control messages we are going to add a second Endpoint for the application. The purpose of the second Endpoint will be to manage a new cluster that will be used for controlling the timeout period of the send timer on the Router. Note that in this case we are going to implement both Endpoints within the same .c and .h file. There is no reason that one could not or should not isolate endpoints to a specific file, or alternatively implement the data and control clusters in the same endpoint. We have done it this way to simplify the logic and take advantage of the existing ZDO support functionality without making any further modifications to the code.

4. New Endpoint and Cluster

In this section we need to add a new Cluster for configuring the report period of the sending device (Router). In the next section we are going to use centralized binding to bind the Router and End Device so that the End Device can control the report period of the Router dynamically.

In GenericApp.h we need to add our new Endpoint ID and a new Cluster for updating the report period of the sender. We will call this a timeout cluster since it sets the timeout (period) in the sending application code that we previously defined.

There’s a lot of typing in this section, so cut/paste from Lab8C_Code.txt in the Source folder of this project to reduce typos.

Add the following definitions in GenericApp.h:

```c
#define GENERICAPP_CTRL_ENDPOINT       0x55
#define GENERICAPP_MAX_CTRL_CLUSTERS  1
#define GENERICAPP_TIMEOUT_CLUSTER    2
```

Like the existing GenericApp endpoint 10, this Endpoint Number and Cluster ID are just picked out of the air. We’ll cover how to obtain real numbers in a later module.

Now add a cluster list for the control clusters that will be used later on when we create a binding table entry locally. Add this code in the Global Variables section of GenericApp.c where the other cluster list is defined.

```c
// This list is for our new timeout control cluster
const cId_t GenericApp_ClusterCtrlList[GENERICAPP_MAX_CTRL_CLUSTERS] =
{      
    GENERICAPP_TIMEOUT_CLUSTER
};
```
5. Descriptors

Next, we are going to need a second Simple Descriptor and Endpoint Descriptor that allow us to differentiate between the existing data cluster (already created and registered with the AF) and a new control cluster which we will also register and use for binding. Similar to what exists for the data Cluster, create a new Simple Descriptor and Endpoint Descriptor to register with the AF. We’ll differentiate between the devices with preprocessor definitions that we’ll add in a moment. Add the code below at the bottom of the Global Variables area in GenericApp.c.

```c
// Simple Descriptor for Control Clusters
c
const SimpleDescriptionFormat_t GenericApp_SimpleCtrlDesc =
{
    GENERICAPP_CTRL_ENDPOINT,               // int Endpoint;
    GENERICAPP_PROFID,                      // uint16 AppProfId[2];
    GENERICAPP_DEVICEID,                   // uint16 AppDeviceId[2];
    GENERICAPP_DEVICE_VERSION,          // int AppDevVer:4;
    GENERICAPP_FLAGS,                       // int AppFlags:4;
#if !defined(COORDINATOR)                   // Router and End Device
    0,                                           // no input clusters
    (cId_t *) NULL,
    GENERICAPP_MAX_CTRL_CLUSTERS,   // byte AppNumOutClusters;
    (cId_t *) GenericApp_ClusterCtrlList,       // byte *pAppOutClusterList;
#endif
#if defined(COORDINATOR)                     // Coordinator
    GENERICAPP_MAX_CTRL_CLUSTERS,   // byte AppNumInClusters;
    (cId_t *) GenericApp_ClusterCtrlList,  // byte *pAppInClusterList;
    0,                                           // no output clusters
    (cId_t *) NULL,
    #endif
};

// Endpoint Descriptor for Control Clusters
#if !defined(COORDINATOR)
endPointDesc_t GenericApp_ctrlEpDesc;
#endif
```

6. Add Preprocessor Definitions

For each Workspace configuration, open the Project Options. Under the C/C++ compiler category, click on the Preprocessor tab. Add COORDINATOR to the Coordinator’s options, ROUTER to the Router options and ENDDEVICE to the End Device options in the Defined symbols list.
7. Initialize and register your new Descriptors

Now that you have defined your new control clusters and descriptors to represent them within the AF, you must populate and register the descriptors with the AF. Do this in GenericApp.c in the GenericApp_Init() function after the last ZDO_RegisterForZDOMsg() call:

```c
// populate and register the new control descriptors with the AF
#if !defined(COORDINATOR)
  GenericApp_ctrlEpDesc.endPoint = GENERICAPP_CTRL_ENDPOINT;
  GenericApp_ctrlEpDesc.task_id = & GenericApp_TaskID;
  GenericApp_ctrlEpDesc.simpleDesc
      = (SimpleDescriptionFormat_t *)& GenericApp_SimpleCtrlDesc;
  GenericApp_ctrlEpDesc.latencyReq = noLatencyReqs;

  afRegister( & GenericApp_ctrlEpDesc );
#endif
```

8. Verify your work so far

At this point we need a sanity check to make sure we are free from compiler/linker errors and that everything still executes as expected. However, we haven’t yet added any new binding or clusters. Since we’re concerned with the binding process, let’s simplify our Sniffer view by temporarily turning off the transmission of data. The sending of data is initiated by the timer set during the handling of the ZDO_STATE_CHANGE event (in the GenericApp_ProcessEvent() function). Comment out the first osal_start_timerEx() call so that we will no longer see periodic traffic.

Build and load the code on the Coordinator, Router and End Device. Run the Packet Sniffer on the appropriate channel, and turn on the Coordinator, then Router, then End Device. Initiate the automated binding process by pressing the joystick left (SW4) on the End Device. Use the Packet Sniffer of your choice to watch the crazy action.
Using Centralized Binding

This mechanism uses a button press or other similar action at the selected devices to bind within a specific timeout period. The End Device Bind Request messages are collected at the Coordinator within the timeout period and a resulting Binding Table entry is created based on the agreement of profile ID and cluster ID. The default end device binding timeout (APS_DEFAULT_MAXBINDING_TIME) is 16 seconds (defined in ZGlobals.h), but can be modified if added to f8wConfig.cfg. The code that is run by pressing SW2 in GenericApp is an example of an End Device Bind implementation. The key handler in GenericApp.c calls ZDP_EndDeviceBindReq() in ZDProfile.c, which gathers all the information for the application’s endpoint and sends it to the Coordinator.

When the Coordinator receives 2 matching End Device Bind Requests within the timeout period, it will start the process of creating source binding entries in the requesting devices. The Coordinator follows the following process, assuming matches were found in the ZDO End Device Bind Requests:

1. Send a ZDO Unbind Request to the first device. The End Device Bind is toggle process, so an unbind is sent first to remove any possible existing bind entry.
2. Wait for the ZDO Unbind Response, if the response status is ZDP_NO_ENTRY, send a ZDO Bind Request to make the binding entry in the source device. If the response status is ZDP_SUCCESS, move on to the cluster ID for the first device (the unbind removed the entry – toggle).
3. Wait for the ZDO Bind Response. When received, move on to the next cluster ID for the first device.
4. When the first device is done, do the same process with the second device.
5. When the second device is done, send the ZDO End Device Bind Response messages to both the first and second device.

9. Enable REFLECTOR option

In order for this process to take place we need to first enable local binding storage by enabling the REFLECTOR compile option. In both the Router and End Device workspace options add the compile option for REFLECTOR if it isn’t already defined. Just in case you need the steps to do this: Right click the Workspace name for the selected project, go to C/C++ Compile → Preprocessor tab, and in the Defined Symbols box add REFLECTOR.
10. **Send Bind Request**

Add code in the SW1 (joystick UP) handling code to invoke the binding process using `ZDP_EndDeviceBindReq()`. Make sure you pass the correct Endpoint ID when invoking this call. The following code will implement centralized binding between our new control endpoints when SW1 is pressed on the Router and End Device. If the binding is successful, `GenericApp_ProcessZDOMsgs()` will light LED4 to indicate success or failure of the process.

Be careful here, it’s easy to confuse the SW1 handling code with the shift-SW1 code that we’ll be using later. LOOK at the code first.

```
#if !defined(COORDINATOR)
    HalLedSet ( HAL_LED_4, HAL_LED_MODE_OFF );
    dstAddr.addrMode = Addr16Bit;
    dstAddr.addr.shortAddr = 0x0000; // Coordinator
    ZDP_EndDeviceBindReq( &dstAddr, NLME_GetShortAddr(),
        GenericApp_ctrlEpDesc.epDesc.endPoint,
        GENERICAPP_PROFID,
        GENERICAPP_MAX_CTRL_CLUSTERS, (cId_t *)GenericApp_ClusterCtrlList,
        GENERICAPP_MAX_CTRL_CLUSTERS, (cId_t *)GenericApp_ClusterCtrlList,
        FALSE );
#endif
```

11. **Build and download the code.**

Follow the normal build/load and start-up process.

Now observe the binding process when you **press SW1** (joystick up) on the **Router** followed by **pressing SW1** on the **End Device** within a 16 second window. Monitor the process on the Dain-tree SNA.

**Note:** Make sure that when initiating binding you start first on the Router and second on the End Device. If you would like to change the order of this process you are going to need to turn periodic polling back on. Try to think about what is going on during the binding process and which layers initiate communication to understand why periodic polling needs to be enabled for it to work both ways. If you experiment with this, you should turn periodic polling off and make sure you initiate End Device Binding on the Router first as the traffic on the **Packet Sniffer** can get confusing later on.

If your brand new binding is successful, LED4 on the Router and End Device will light. There won’t be any “Hello World” messages since we disabled the OSAL timer (and we didn’t bind that application by pressing SW2). Sweet!

We’ll save the rest of the steps for configuring the timing interval remotely until the next lab.

Now, let’s add some instrumentation, using Z-Tool, so we know what’s going on in the code.
Z-Tool

Z-Tool provides the programmer with insight into the code operation by sending code initiated messages out through the UART port as well as allowing data and commands to be received by the code. Let’s use the tool to instrument the existing code.

12. Set Options

In the End Device Project Options, select the C/C++ Compiler category and click on the Preprocessor tab. Verify that the following Defined symbols are present:

- ZTOOL_P1 (Use UART1 on the SmartRF05)
- MT_TASK (Add basic Monitor - Test functionality)
- MT_SYS_FUNC (Add SYS Monitor – Test support)
- MT_ZDO_FUNC (Add ZDO Monitor – Test support)
- MT_APP_FUNC (Add APP Monitor - Test support)

A quick look in the documents folder at the Z-Stack Compile options document will provide some additional explanation.

Adding this functionality increases the code size, so go to the Router and Coordinator defined symbols and remove the ZTOOL_P1 and other other MT type symbols from the list, if they are present.

13. Add MT Header includes

Add these two MT header file includes at the beginning of GenericApp.c at the bottom of the include area.

```c
/* MEASUREMENT and TEST */
#include "MT.h"
#include "MT_APP.h"
```

14. Re-enable Periodic Data Transmission

Let’s make the code fully operational again. Back in step 8, we commented out the first osal_start_timerEx() statement. Uncomment it now.

15. Add some Instrumentation to the Code

Let’s add some instrumentation, just like you would add a printf() to test a section of code. At the same time, let’s see how well you’ve learned how GenericApp.c works. Insert the code below, replacing the question marks with the text shown, in the following places.

- “Network Join Success” (after the first timer start)
- “Hello World Message Sent” (after the second timer start)
- “SW1 pressed” (below the bind request)
- “Binding Successful” (inside the ZSuccess if construct, right after the LED is lit)

```c
#if defined( MT_TASK )
    debug_str( "?????" );
#endif
```
15. Build and Download

If we’ve done our if/defs correctly, we should only need to rebuild the End Device right now. Build and download the project to the board, then click Stop Debugging and minimize IAR Embedded Workbench. Power down the board. Connect the RS-232 cable from the serial port connection on the back of the PC to the serial connector on the End Device SmartRF05 board. Power up the boards in the normal manner.

15. Z-Tool

Start Z-Tool from the desktop shortcut. The tool will automatically scan the available COM ports for the MT enabled EB board.

Once Z-Tool is running, you will notice three directories of MT commands on the left menu:

- System – System level general commands (compiled in by MT_TASK)
- ZDO – The MT interface to the ZDO for management and control
- App – For sending user configurable messages to the Stack

In Z-Tool, you will see that the OSAL timer is running, sending out the “Hello World” message, even though no binding has taken place. We missed the initial messages, so power down the End Device board, clear the Z-Tool display and power the board back on. After the first “Hello World” message in Z-Tool, press SW1 (joystick UP) first on the Router and then on the End Device to bind the applications. Power the board off to stop the message traffic. In Z-Tool you should see:

- A reset response
- The board’s IEEE address
- Association confirmation
- Device type, short address and parent identification
- Network joining success
- Hello World message sent
- SW1 pressed
- Binding Successful

The first four messages are automatically sent when MT_TASK is enabled. Hopefully, the rest of your instrumentation worked perfectly.

The way we’ve coded things, we just bound the new endpoints on the End Device and Router. We can also bind the old “Hello World” endpoints by pressing SW2 (joystick RIGHT) on the Router, then the End Device. Do that now and watch the “Hello World” messages appear on the Router LCD.

16. Check out some other cool Z-Tool tricks

In the left hand Z-Tool window, open the System group and find SYS_ADC_READ. Channel 0 of this ADC is connected to the potentiometer on the SmartRF05 board. Click on SYS_ADC_READ and change Resolution in the Message Parameters to 3 (12-bit). Rotate the pot fully counter-clockwise, right click on SYS_ADC_READ and click Send Message. You should read 0x0000 in the Z-Tool response. Rotate the pot clockwise and send the message again.

Try some of the other command and see what happens.

17. Close all open programs and shut down the boards. You’re done.
ZDO APIs

ZDO APIs provide application level control and monitoring of the following services through the ZigBee Device Profile (ZDP):

- **Device and Service Discovery**
  - Ability to discover services offered by other network devices
- **End Device Bind, Bind and Unbind Service**
  - Creation and deletion of binding table entries mapping messages to their destination
- **Network Management Service**
  - Allows user or commissioning tools to manage the network
- **Device Network Startup**
  - ZDApp_Init() in ZDApp.c provides network startup by default
  - Your application can override this behavior by including the HOLD_AUTO_START compile option (NV_RESTORE also recommended)

ZDO Network Startup

```c
void ZDApp_Init( uint8 task_id )
{
    ZDAppTaskID = task_id;
    ZDAppNwkAddr.addrMode = Addr16Bit;
    ZDAppNwkAddr.addr.shortAddr = INVALID_NODE_ADDR;
    (void)NLME_GetExtAddr();
    ZDAppCheckForHoldKey(); // Check for manual "Hold Auto Start"
    ZDO_Init();
    afRegister( (endPointDesc_t *)&ZDApp_epDesc );
    #if defined( ZDO_USERDESC_RESPONSE )
        ZDApp_InitUserDesc();
    #endif // ZDO_USERDESC_RESPONSE
    // Start the device?
    if ( devState != DEV_HOLD )
    { ZDOInitDevice( 0 ); } else
    { HalLedBlink ( HAL_LED_4, 0, 50, 500 );
        ZDApp_RegisterCBs();
        ZDApp_Init() /* ZDApp_Init() */
    }
```

- This is the default network start code (minus some comments)
- Initiates network
- Registers endpoint
- Initiates device (NV_RESTORE optional)
- Registers for ZDO callbacks based on compiler options set and device type (C/R/ED)
ZDO Status Indicator

devState provides the user app with the ZDO status:

typedef enum
{
    DEV_HOLD,               // Initialized - not started automatically
    DEV_INIT,              // Initialized - not connected to anything
    DEV_NWK_DISC,           // Discovering PAN's to join
    DEV_NWK_JOINING,        // Joining a PAN
    DEV_NWK_REJOIN,         // Rejoining a PAN, only for end devices
    DEV_END_DEVICE_UNAUTH,  // Joined but not yet authenticated by trust center
    DEV_END_DEVICE,         // Started as device after authentication
    DEV_ROUTER,             // Device joined, authenticated and is a router
    DEV_COORD_STARTING,     // Started as Zigbee Coordinator
    DEV_ZB_COORD,           // Started as Zigbee Coordinator
    DEV_NWK_ORPHAN          // Device has lost information about its parent.
} devStates_t

Registering for a ZDO Callback

- You can register for ZDO over-the-air (OTA) message (request or receive) callbacks
- These messages would normally be transparent to your application
  
  ZDO_RegisterForZDOMsg()

- Message is sent to the application as an OSAL message (ZDO_CB_MSG)
- zdoIncomingMsg_t contains the OTA message body
### Mobile End Devices

- End devices are automatically mobile. No extra compile flags needed
- An end device detects that a parent isn’t responding either through polling failures and/or through data message failures
- `MAX_POLL_FAILURE_RETRIES` controls the number of consecutive failures. See `FwConfig.cfg` (higher = less sensitive)
- When a child network layer detects that its parent isn’t responding, it will initiate a rejoin by calling `ZDO_SyncIndicationCB()`
- The rejoin process will first scan for an existing parent, then scan for a new parent and join that network
- In a secure network, it’s assumed that the device already has a key and a new key isn’t issued

Let's do a lab...
*** there’s no future in acting your age ***
Lab 8D – Send and Receive Control Cluster

- Add Timeout cluster send code
- Add Control key handler s/w
- Add Timeout receive code
- Test and verify

<table>
<thead>
<tr>
<th>Group</th>
<th>Channel</th>
<th>PAN ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x0C (12)</td>
<td>0x0AAA</td>
</tr>
<tr>
<td>2</td>
<td>0x0D (13)</td>
<td>0x0BEE</td>
</tr>
<tr>
<td>3</td>
<td>0x0E (14)</td>
<td>0x0CEE</td>
</tr>
<tr>
<td>4</td>
<td>0x0F (15)</td>
<td>0x0DEE</td>
</tr>
<tr>
<td>5</td>
<td>0x10 (16)</td>
<td>0x0EEE</td>
</tr>
<tr>
<td>6</td>
<td>0x11 (17)</td>
<td>0x0EFF</td>
</tr>
<tr>
<td>7</td>
<td>0x12 (18)</td>
<td>0x0BAB</td>
</tr>
<tr>
<td>8</td>
<td>0x13 (19)</td>
<td>0x0ACE</td>
</tr>
</tbody>
</table>
Hardware list:

- 3 SmartRF05EB boards
- 3 CCMSP-EM430F2618 boards
- 3 CC2520EM boards
- 1 SmartRF04EB board
- 1 CC2430EM board
- 4 antennas
- MSP-FET430UIF and ribbon cable
- 5 USB A/B cables
- 2 AA batteries
- Post-it™ flags

Software list:

- IAR Embedded Workbench for MSP430 version 4.11D
- TI Packet Sniffer version 2.10.1
- Daintree Sensor Network Analyzer version 2.3.0.8
- Z-Tool version 2.0

(You will find shortcuts for the above applications on the desktop)

- Z-Stack version 2.1.0 – 1.3.0
Procedure

Controlling the Router from the End Device

Now that we have our binding established between the End Device and the Router we will add code that utilizes this binding to send a message to the Router to adjust its reporting period. Note that this code will be applicable to the End Device only.

1. Start IAR Embedded Workbench

Start up IAR Embedded Workbench and open the GenericApp.eww project located in: C:\Texas Instruments\ZStack-2.1.0-1.3.0\Projects\zstack\Samples\Lab8D\CC2520DB. This is a copy of the solution from Lab8C.

Open GenericApp.c and Lab8D_Code.txt for editing.

2. Sending Control Information

Sending data to a device requires a call to the AF_DataRequest() function, so we need to create a second local function for sending this new message.

Add the following local function declaration in GenericApp.c. As before, you can cut/paste from Lab8D_Code.txt if you like.

```
void GenericApp_SendTheControl( uint16 reportPeriod );
```
Add GenericApp_SendTheControl() function

Add the following GenericApp_SendTheControl() function to the end of GenericApp.c.

```c
#ifndef defined(ENDDEVICE)
/*
 * @fn       GenericApp_SendTheControl
 * @brief    Send a control message.
 * @param    timeout
 * @return   none
 */
void GenericApp_SendTheControl( uint16 reportPeriod )
{
    afAddrType_t addrPlaceholder;
    addrPlaceholder.addrMode = afAddrNotPresent;

    if ( AF_DataRequest( &addrPlaceholder, &GenericApp_ctrlEpDesc,
                         GENERICAPP_TIMEOUT_CLUSTPR,
                         (byte)sizeof(reportPeriod),
                         (byte *)&reportPeriod,
                         &GenericApp_TransID,
                         AF_DISCV_ROUTE | AF_ACK_REQUEST,
                         AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
    {
        // Successfully requested to be sent.
    }
    else
    {
        // Error occurred in request to send.
    }
}
#endif
```

addrPlaceHolder is a place holder for the destination address. Using afAddrNotPresent in the address mode will initiate a look-up in the binding table for the address bound to the sending application. The stack software can then utilize that address to send the message to the correct destination.

Look at the function AF_DataRequest() defined in AF.c if you are curious about exactly how this works.
4. Add Key Handler Code

Add the code to send control information to the key handler code. Note that the if statements are already in the existing code. We’re using the shift key (BUTTON1 - the red button at the bottom right corner of the board) so you don’t modify the reporting period by accident. Add code for each switch press to be a different interval (1000, 2000, 5000, 10000).

```c
if ( shift )
{
    if ( keys & HAL_KEY_SW_1 )
    {
#if defined (ENDDEVICE)
        GenericApp_SendTheControl( 1000 );
#endif
    }
    if ( keys & HAL_KEY_SW_2 )
    {
#if defined (ENDDEVICE)
        GenericApp_SendTheControl ( 2000 );
#endif
    }
    if ( keys & HAL_KEY_SW_3 )
    {
#if defined (ENDDEVICE)
        GenericApp_SendTheControl ( 5000 );
#endif
    }
    if ( keys & HAL_KEY_SW_4 )
    {
#if defined ENDDEVICE)
        GenericApp_SendTheControl ( 10000 );
#endif
    }
}
```

5. Build, Download and Test

Build, download and run the code to verify that you have implemented everything correctly. You only need to build/download the End Device configuration at this time. While we have not added code on the Router to handle the new configuration messages, the existing code should allow us to create the binding and send the update packet. Catch the wild action on the Daintree Packet Sniffer and note the Cluster ID and Endpoint ID, as well as other application headers.

Some reminders: The shift key on the SmartRF05 board is the red button (BUTTON1) nearest to the RS-232 port. Remember that you must bind the End Device to the Router first, by pressing SW1 (joystick UP) on the Router and then pressing SW1 on the End Device. Also, remember that the End Device is the one sending the control packet, so you should hold BUTTON1 and toggle the joystick on the End Device to send the control messages.
Message Timing

6. Updating the Router’s Timer

Now that our End Device is bound to our Router with respect to the timing control message, and
we have demonstrated that we can successfully send this control message based on the shift-
switch press, we must now add the code to handle this message on the Router.

The application receives notification of an incoming message through a system callback
(AF_INCOMING_MSG_CMD). Upon receiving this callback, our application calls
GenericApp_MessageMSGCB() to parse and handle this message. Within this function we ex-
amine the ClusterId field of the incoming packet to determine the type of message we received.

7. Handling the New Control Cluster

Add the necessary code in GenericApp_MessageMSGCB() to handle the new Configuration
Cluster you created, and update your application variable that controls the timing between outgo-
ing messages based on the received message. You can use the code from the same function that
extracts the CLUSTERID payload as reference. Although it is not necessary, you may surround
your new case statement with pre-processor directives to compile out the code if the device is
programmed as the Coordinator (ZDO_COORDINATOR) or End Device. However this
should not be necessary since a Coordinator or End Device will never receive this message (as-
suming everything up until this point was done correctly of course).

```c
void GenericApp_MessageMSGCB( afIncomingMSGPacket_t *pkt )
{
    switch ( pkt->clusterId )
    {
        case GENERICAPP_CLUSTERID:
            // "the" message
            #if defined( LCD_SUPPORTED )
                HalLcdWriteString( (char*)pkt->cmd.Data, HAL_LCD_LINE_1 );
                Count++;
                HalLcdWriteValue( count, 10, HAL_LCD_LINE_3 );
            #elif defined( WIN32 )
                WPRINTSTR( pkt->cmd.Data );
            #endif
            break;
        #if !defined (COORDINATOR)
            case GENERICAPP_TIMEOUT_CLUSTER:
                // update timeout variable for outgoing "CLUSTERID" message
                SendMsgTimeout = *((uint16 *)(pkt->cmd.Data));
                break;
        #endif
    }
}
```
8. Build, Download and Test

Build and download your project to all three devices. Verify that your changes worked as expected and that the End Device is now capable of dynamically changing the report time of the Router.

Things can get a little confusing here, so let’s step through the process in the following manner while observing the Daintree SNA. Do it CAREFULLY or it won’t work.

1. Power on the Coordinator
2. Power on the Router
3. Press SW4 (joystick left) on the Router to initiate automatch binding with the Coordinator. The Router will begin sending messages to the Coordinator at the default period of 5 seconds. Observe the Coordinator’s LCD display.
4. Power on the End Device
5. Press SW1 (joystick up) on the Router, then the End Device within 16 seconds to initiate centralized binding for the control application.
6. Press and hold BUTTON1 and then press SW1 (joystick up) on the End Device. This will send a control message over the air and changed the reporting period to 1 second. If you watch the Coordinator’s LCD or the Daintree SNA display, you can see this rate change. Try the other periods:

   • SW1 (joystick up) = 1 second
   • SW2 (joystick right) = 2 seconds
   • SW3 (joystick down) = 5 seconds
   • SW4 (joystick left) = 10 seconds

Notice that each time a message is sent, the green LED on the Sniffer board lights. When you send a message (in step 6), the LED should light. If it does not, you’ve messed up. Double-check your steps.

So let’s go over what we’ve accomplished here. The End Device and Coordinator have bound Endpoints that allow the End Device to send a message containing the timer delay (1000, 2000, 5000, 10,000). The code on the Router then programs a timer with that delay to send out a “Hello World” message. The Coordinator and Router have bound Endpoints that allows the Coordinator to display the message on it’s LCD. Overall, that’s a pretty comprehensive overview of the Z-Stack capabilities …

9. You’re finished

Shut down all running software and boards.

You’re done.
Smart Energy

ZigBee Smart Energy

Why Smart Energy?

- Policies such as the Energy Policy Act of 2005 and California’s Title 24
  - Require utility companies to adopt “green” technologies
  - Promote awareness of energy conservation
  - Push standardization of wireless protocols such as ZigBee's Smart Energy
  - Help establish National and International market acceptance
  - Drive trial deployments to prove feasibility early
What is “Smart Energy”

- **Metering**
  - Multiple commodities including electric, gas, water, and thermal
  - Real-time consumption / production information
  - Historical price information
  - Support for meter-as-gateway and meter-as-device on a ZigBee network
- **Demand Response and Load Control**
  - Scheduling of multiple events
  - Built in support for customer override
- **Text Messaging**
- **Security**
  - Elliptic curve support via 3rd party library

TI’s Smart Energy Solution

- TI / Chipcon has been driving the Alliance since inception
- TI has implemented the full Smart Energy Profile
  - Available free for download today
  - Supports SETestApp Sample Application Code
  - Supports Elliptic Curve Cryptology (ECC) stubs
- TI can deliver quality parts of significant volume in a timely manner
- TI is well positioned in ZigBee, 6LoWPAN, or wherever the Metering market takes us
- TI supports Over the Air Download for remote firmware upgrades
- TI will release an SoC based complete Smart Energy solution for 2009
**Elliptic Curve Cryptology (ECC)**

- TI provides the code for hooking Certicom's ECC solution into our stack. We do not provide hooks for any other ECC solution.
- TI does NOT provide the ECC code itself.
- Certicom charges customers for their development kit and does not provide ECC for free. Please check with Certicom for pricing.

www.Certicom.com
Demos

- OAD
- Commissioning
- Mobility

Large Network Demo

- Use Lab8D software
- Set Channel = 0xF (15) and PAN ID = 0xFFFF
- Program a single board as a Coordinator
- Program 7 boards as Routers … Battery powered
- Remainder of boards as End Devices … Battery powered
- No formal steps … play! Observe traffic on the Daintree SNA
Demos

Texas Instruments