ember

EmberZNet Application Developer's Guide

14 November 2008

120-4028-000D

Ember Corporation
47 Farnsworth Street
Boston, MA 02210
+1 (617) 951-0200
0Hwww.ember.com

wireless semiconductor solutions

Ember Corporation
47 Farnsworth Street
Boston, MA 02210
+1 (617) 951-0200
1Hwww.ember.com

Contents

1 About this Guide

Purpose

This document describes how you can use the EmberZNet stack to create an application. It provides a basic introduction to ZigBee, EmberZNet's components and capabilities, and a walk through of applications Ember provides that you can modify to create a custom ZigBee application. You can read this entire document to learn about these EmberZNet applications or just refer to specific chapters as necessary.

Audience

This document is intended for project managers and for embedded software engineers who need to build a ZigBee application using EmberZNet, and assumes that the reader has a solid understanding of embedded systems design and programming in the C language. Experience with networking and radio frequency systems is useful but not required. For a more in-depth review of specific topics, please refer to the *EmberZNet Application Developer's Reference Manual* (120-3021-000).

Document Organization

Chapter 2 provides an overview of the EmberZNet API, and Chapter 3 describes how to use the Hardware Abstraction Layer for a new developer on EmberZNet. The rest of the chapters introduce the sample applications Ember provides:

- Chapter 4: Home Automation application
- Chapter 5: Sensor Sink application
- Chapter 6: Rangetest application
- Chapter 7: Training application
- Chapter 8: ZDO application

You can use the source code provided for these applications as a starting point for your own application development.

If you have questions after reviewing any of the sample applications, please contact support@ember.com for assistance.

2 Introducing EmberZNet API

This chapter introduces the EmberZNet API. The EmberZNet API controls the EmberZNet stack library and provides function calls and callbacks related to the network formation, discovery, joining, and messaging capabilities. For full reference documentation of the functions and their parameters, see the *EmberZNet API Reference* (120-3013-000).

Ember recommends that software engineers new to EmberZNet or those who are looking to refresh their understanding of the different components of the API read this chapter. You will see how the API can help you to quickly develop applications.

API Organization

To make the API more manageable, it is broken into 15 functional sections. This chapter provides a detailed introduction to 6 of the fundamental API sections:

- Network Stack Usage
- Network Formation
- Sending Messages and Receiving Messages
- Packet Buffers
- Security and Trust Center
- End Devices
- Event Scheduling

The other functional sections are:

- Binding Table
- Stack Information
- Ember Common Data Types
- Configuration, Status Codes
- Stack Tokens
- ZDO
- Bootloader
- Manufacturing and Functional Test Library
- Debugging Utilities

Naming Conventions

All functions that are part of the public EmberZNet API begin with the prefix `ember`. Ember strongly recommends that you maintain this convention when writing custom software so that it is easy to find information and documentation pertaining to a function.

API Files and Directory Structure

The following list describes files within the stack that contain useful information.

- **<stack>/config/config.h:** This file contains the stack build revision and can be used when communicating with Ember technical support or when verifying that the stack version used is correct. The format of the version number is described in the file.
- **<stack>/config/ember-configuration-defaults.h:** This file describes compile-time configurable options that affect the behavior of the EmberZNet stack. These should be set in the CONFIGURATION_HEADER or in the Project so that the values are properly set in all files.
- **<stack>/include:** This directory contains all the API header files. The correct ones for the application are included in ember.h, so the application usually only needs to include ember.h. The files can be useful as a reference source for advanced developers. The API reference documentation is generated from these header files.

Network Stack Usage

Ember provides a set of APIs you can use to initialize and operate the Ember network stack.

This section describes `emberNetworkInit()` and `emberInit()`, which initialize the EmberZNet stack upon reboot, and `emberTick()`, which should be called regularly by the application to allow the EmberZNet stack to perform basic tasks like message routing and network maintenance.

Initializing the Network Stack

EmberZNet stack is initialized by calling `emberInit()` in the `main()` function. It may be passed a value for the reset code that can be used for debugging if the device is attached to an InSight Adapter with InSight Desktop.

```
status = emberInit(reset);
```

Note: `emberInit()` must be called before any other stack APIs are used, or the results will be undefined.

For more information about debugging, see the *InSight Desktop User's Guide* (120-4005-000).

Calling `emberNetworkInit()` causes the device to rejoin the network that it was joined to before it rebooted. This will maintain as many of the previous network settings as possible (for example, the network address will be maintained if possible).

```
if (emberNetworkInit() == EMBER_SUCCESS)) {
// Successfully rejoined previous network
} else {
// No previous network or could not successfully rejoin
}
```

Note: On development systems or systems that change device type (both ZR and ZED, for example), the application should verify if the cached device type is the desired device type. This behavior is shown in the sample applications later in this book.

Network Operation

Proper operation of the network is facilitated by calling `emberTick()` regularly in your program loop. The watchdog should also be reset:

```
while(TRUE) {
halResetWatchdog();
emberTick();

// Application-specific functions here
}
```

Network Formation

Functions for creating, joining, and leaving a network have descriptive names: `emberFormNetwork()`, `emberPermitJoining()`, `emberJoinNetwork()`, `emberFindAndRejoinNetwork()`, and `emberLeaveNetwork()`.

Functions for finding a network or determining background energy levels include: `emberStartScan()`, `emberStopScan()`, `emberScanCompleteHandler()`, `emberEnergyScanResultHandler()`, and `emberNetworkFoundHandler()`.

Ember provides wrapper utility functions that hide much of the details of standard network formation:

```
            // Use a function from app/util/common/form-and-join.c
            // that scans and selects a quiet channel to form on.
            // The short PAN ID is randomly picked and the Extended PAN ID is
            // either the one passed in by the app, or (if the app passes 0)
            // is also randomly picked.
            formZigbeeNetwork(EMBER_ALL_802_15_4_CHANNELS_MASK, -1, (int8u*)
      &extendedPanId);
```

This utility function uses `emberStartScan()`, `emberStopScan()`, `emberScanCompleteHandler()`, `emberEnergyScanResultHandler()`, and `emberNetworkFoundHandler()` to discover other networks or determine the background noise level. It then uses `emberFormNetwork()` to create a new network with a unique PAN-ID on a channel with low background noise. Further details can be found in /stack/include/network-formation.h as well as in /app/utils/common/form-and-join.h, and in /app/utils/scan/scan-utils.h for related utilities.

Note: EmberZNet does not have different stack libraries for ZC and ZR devices, so any device that calls `emberFormNetwork()` creates the network and becomes the ZC. As such, only the device starting the network should call `emberFormNetwork()`, and other devices should call `emberJoinNetwork()`, which is described below.

The ZC can then use `emberPermitJoining()` to allow joining, subject to the configured security settings:

```
      emberPermitJoining(60);   // Permit joining for 60 seconds
      emberPermitJoining(0xFF); // Permit joining until turned off
      emberPermitJoining(0);    // Do not permit joining
```

For more information on security settings and authorization, please refer to the security chapter of the *EmberZNet Application Developer's Reference Manual* (document number 120-3021-000).

Joining a Network

Joining a network is accomplished with the `emberJoinNetwork()` API:

```
            status = emberJoinNetwork(EMBER_ROUTER, &networkParams); // To
      join as a ZR
            status = emberJoinNetwork(EMBER_SLEEPY_ZED, &networkParams); //
      To join as a Sleepy ZED
            status = emberJoinNetwork(EMBER_MOBILE_ZED, &networkParams); //
      To join as a Mobile ZED
```

The networkParams variable is a structure of type EmberNetworkParameters and configures the PAN-ID, extended PAN-ID (or 0 for any), channel of the network to join, and the desired TX power with which to join the network.

Ember also provides a utility function that uses `emberStartScan()`, `emberStopScan()`, and `emberScanCompleteHandler()` to discover networks that match the provided options and to join the first one that it finds:

```
// Use a function from app/util/common/form-and-join.c
// that scans and selects a beacon that has:
// 1) allow join=TRUE
// 2) matches the stack profile that the app is using
// 3) matches the extended PAN ID passed in unless "0" is passed
// Once a beacon match is found, emberJoinNetwork is called.
   joinZigbeeNetwork(EMBER_ROUTER, EMBER_ALL_802_15_4_CHANNELS_MASK,
                     -1, (int8u*) extendedPanId);
```

The utility `emberFindandRejoinNetwork()` is used on devices that have lost contact with their network and need to scan and rejoin.

Packet Buffers

The Ember stack provides a full set of functions for managing memory. This memory is statically allocated at link time, but dynamically used during run time. This is a valuable mechanism because it allows you to use statically linked, fixed-length buffers for variable-length messages. This also gives you a better idea of how much RAM your software will require during run time.

Common functions include allocation of buffers with predefined content, copying to/from existing buffers, and freeing allocated buffers. A typical procedure to complete buffer usage is:

1. Allocate a new buffer large enough for length bytes, copy length bytes from dataArray, and check to see that the allocation succeeded:

   ```
   buffer = emberFillLinkedBuffers(dataArray,  length);
   if (buffer == EMBER_NULL_MESSAGE_BUFFER) {
       // the allocation failed!  Do not proceed!
   }
   ```

2. Copy length bytes from buffer into dataArray, starting at index `0`:

   ```
   emberCopyFromLinkedBuffers(buffer, 0, dataArray, length);
   ```

3. Return all memory used by buffer so it can be re-used:

   ```
   emberReleaseMessageBuffer(buffer);
   ```

The following standard memory management and copying functions are available:

`emberAllocateStackBuffers()`
`emberFillStackBuffer()`
`emberFillLinkedBuffers()`
`emberCopyToLinkedBuffer()`
`emberAppendToLinkedBuffers()`
`emberAppendPgmToLinkedBuffers`
`emberAppendPgmStringToLinkedBuffers()`
`emberSetLinkedBuffersLength()`
`emberGetLinkedBuffersByte()`
`emberGetLinkedBuffersPointer()`
`emberSetLinkedBuffersByte()`
`emberCopyLinkedBufferData()`
`emberHoldMessageBuffer()`
`emberMessageBufferContents()`
`emberMessageBufferLength()`
`emberReleaseMessageBuffer()`
`emberSetMessageBufferLength()`

Stack buffers, linked buffers, and message buffers all refer to the same type of data structure. The naming varies depending on the expected usage of the individual functions. Refer to packet-buffer.h for more details.

Address Table or Binding Table Management

The address table is maintained by the network stack and contains IEEE addresses and network short addresses of other devices in the network. Messages can be sent using the address table by specifying the type as `EMBER_OUTGOING_VIA_ADDRESS_TABLE` in commands such as `emberSendUnicast()`. More details on the address table are in message.h.

The binding table can also be used for sending messages. The binding code is within a library, so flash space is not used if the application does not use binding. Refer to binding-table.h for more details.

Sending Messages

Sending messages is simple:

```
// To send to a device previously entered in the address table:
status = emberSendUnicast(EMBER_OUTGOING_VIA_ADDRESS_TABLE,
                          destinationAddressTableIndex,
                          &apsFrame,
                          buffer, &sequenceNum);
// To send to a device via its 16-bit address (if known):
status = emberSendUnicast(EMBER_OUTGOING_DIRECT,
                          destinationId,
                          &apsFrame,
                          buffer, &sequenceNum);
```

In both cases the apsFrame contains the unicast message options, such as retry or enable route discovery, the buffer contains the message, and the sequence number argument provides a pointer to the APS sequence number returned by the stack when the message is queued. In the case of `EMBER_OUTGOING_VIA_ADDRESS_TABLE`, the `destinationAddressTableIndex` should contain the index of the previously stored address table entry.

Broadcast messages are sent in a similar way:

```
// To send a broadcast message:
status = emberSendBroadcast(DESTINATION  //one of 3 ZigBee broadcast
addresses
                            &apsFrame,
                            radius,    // 0 for EMBER_MAX_HOPS
                             buffer, &sequenceNum);
```

The return code should always be checked to see if the stack will attempt delivery.

Refer to message.h for more details on sending or receiving messages.

Note: An `EMBER_SUCCESS` return code does NOT mean that the message was successfully delivered; it only means that the EmberZNet stack has accepted the message for delivery. If `RETRY` is specified on a unicast message, `emberMessageSentHandler()` will be called to inform the application about the delivery results.

Receiving Messages

Incoming messages are received through the `emberIncomingMessageHandler()`, a handler function that is called by the EmberZNet stack and implemented by the application. The parameters passed to the function are:

- Message Type: for example, UNICAST, BROADCAST
- APS Frame
- Message buffer containing the data contents of the message

Several functions are only available within the context of the `emberIncomingMessageHandler()` function:

- **`emberGetLastHopLqi()`**: return the incoming LQI of the last hop transmission of this message
- **`emberGetLastHopRssi()`**: return the incoming RSSI of the last hop transmission of this message
- **`emberGetSender()`**: get the sender's 16-bit network address
- **`emberGetSenderEui64()`**: get the sender's 64-bit IEEE address

  Note: This is available only if the sender included the 64-bit address—see the API reference for more information.

- **`emberSendReply()`** allows a message to be sent in reply to an incoming unicast message.

Source Routes and Large Networks

Aggregation routes (also called "many-to-one routes") are used to efficiently create network-wide routes to the gateway device(s). Source routes are then used from these gateway devices to send messages back to devices in the network. The source route is specified in the message network header, reducing the route-related memory requirements on intermediate devices. The functions `emberSendManyToOneRouteRequest()`, `emberAppendSourceRouteHandler()`, `emberIncomingRouteRecordHandler()`, `emberIncomingManyToOneRouteRequestHandler()`, `emberIncomingRouteErrorHandler()` are all used during source routing.

Security and Trust Center

Security policies for the network are established by the trust center when the network is formed. Devices joining a network must use the existing security policies or they will not be allowed to join. See the *EmberZNet Application Developer's Reference Manual* (120-3021-000) for a detailed discussion of ZigBee and EmberZNet security settings. Details are also included in /stack/include/security.h.

End Devices

EmberZNet provides two types of end devices, Sleepy End Devices (Sleepy ZED) and Mobile End Devices (Mobile ZED). Mobile ZEDs are expected to move, so information on these devices is not saved in parent devices. Sleepy end devices are expected to maintain the same parent device except in cases where the parent is lost.

For ZEDs, the APIs provide sleep and wake, parent polling, and parent status functions. For parent routers (including the coordinator), the APIs provide child polling event notification and child management functionality.

Refer to child.h for more details on these functions.

Event Scheduling

The Event Scheduling macros implement an event abstraction that allows the application to schedule code to run after some specified time interval. Events are also useful for when an ISR needs to initiate an action that should run outside of the ISR context.

While custom event-handling code can be written by the application, Ember recommends that developers consider using this system first before consuming additional flash and RAM, which duplicates its functionality. Refer to event.h for more details.

3 Introducing the Ember HAL

The Ember Hardware Abstraction Layer (HAL) is program code between a system's hardware and its software that provides a consistent interface for applications that can run on several different hardware platforms. To take advantage of this capability, applications should access hardware through the API provided by the HAL, rather than directly. Then, when you move to new hardware, you only need to update the HAL. In some cases, due to extreme differences in hardware, the HAL API may also change slightly to accommodate the new hardware. In these cases, the limited scope of the update makes moving the application easier with the HAL than without.

The introductory parts of this chapter are recommended for all software developers who are using EmberZNet. Developers needing to modify the HAL or port it to new a hardware platform will want to read the entire chapter to understand how to make changes while meeting the requirements of the EmberZNet stack.

HAL API Organization

The HAL API is organized into the following functional sections:

- Common microcontroller functions**:** APIs for control of the MCU behavior and configuration.
- Token access**:** EEPROM, Simulated EEPROM (SimEEPROM), and Token abstraction. For a detailed discussion of the token system, see the *EmberZNet Application Developer's Reference Manual* (120-3021-000).
- Peripheral access**:** APIs for controlling and accessing system peripherals.
- System timer control**:** APIs for controlling and accessing the system timers.
- Bootloading**:** The use of bootloading is covered in the Bootloading chapter of the *EmberZNet Application Developer's Reference Manual* (120-3021-000).
- HAL utilities**:** General-purpose APIs that may rely on hardware capabilities (for example, CRC calculation that may take advantage of hardware acceleration).

Naming Conventions

HAL function names have the following prefix conventions:

- `hal`**:** The API sample applications use. You can remove or change the implementations of these functions as needed.
- `halCommon`**:** The API used by the EmberZNet stack and that can also be called from an application. Custom HAL modifications must maintain the functionality of these functions.
- `halStack`**:** Only the EmberZNet stack uses this API. **These functions should not be directly called from any application**, as this may violate timing constraints or cause re-entrancy problems. Custom HAL modifications must maintain the functionality of these functions.
- `halInternal`**:** The API that is internal to the HAL. **These functions are not called directly from the stack and should not be called directly from any application.** They are called only from

halStack or halCommon functions. You can modify these functions, but be careful to maintain the proper functionality of any dependent halStack or halCommon functions.

Most applications will call *halXXX* and *halCommonXXX* functions and will not need to modify the HAL. If you need a special implementation or modification of the HAL, be sure to read the rest of this chapter as well as the datasheet for your Ember platform first.

API Files and Directory Structure

The HAL directory structure and files are organized to facilitate independent modification of the compiler, the MCU, and the PCB configuration.

- **<hal>/hal.h:** This master include file comprises all other relevant HAL include files, and you should include it in any source file that uses the HAL functionality. **Most programs should not include lower-level includes**, and instead should include this top-level hal.h.
- **<hal>/ember-configuration.c:** This file defines the storage for compile-time configurable stack variables and implements default implementations of functions. You can customize many of these functions by defining a preprocessor variable at compile-time and implementing a custom version of the function in the application. (For more information, see ember-configuration-defaults.h in the *EmberZNet API Reference* (120-3016-000.)
- **<hal>/micro/generic:** This directory contains files used for general MCUs on POSIX-compliant systems. The default compiler is GCC.
- EM250 HAL implementation
  - **<hal>/micro/xap2b:** This directory contains the implementation of the HAL for the XAP2b, which is the processor core used by the EM250. Functions in this directory are specific to the XAP2b but are not specific to the EM250 (see the next entry).
  - **<hal>/micro/xap2b/em250:** This directory implements functions that are specific to the EM250.
  - **<hal>/micro/xap2b/em250/board:** This directory contains header files that define the peripheral configuration and other PCB-level settings, such as initialization functions. These are used in the HAL implementations to provide the correct configurations for different PCBs.
- EM2420 HAL implementation
  - **<hal>/micro/avr-atmega/128:** This directory contains the implementation of the HAL for the Atmel AVR 128 microprocessor. Functions in this directory are specific to the AVR 128. Note that these directories are only included in EM2420 software releases.

HAL API Description

This section gives an overview of each of the main subsections of the HAL functionality.

Common microcontroller functions

Common microcontroller functions include halInit(), halSleep(), and halReboot(). Most applications will only need to call halInit(), halSleep() (usually only ZEDs), and halResetWatchdog(). The functions halInit(), halSleep(), halPowerUp(), halPowerDown(), and so on call the proper functions defined in the board header file to initialize or power down any board-level peripherals.

Token access

EmberZNet uses persistent storage to maintain manufacturing and network configuration information when power is lost or the device is rebooted. This data is stored in *tokens*. A token consists of two parts: a key used to map to the physical location, and data associated with that key. Using this key-

based system hides the data's location from the application, which allows support for different storage mechanisms and the use of flash wear-leveling algorithms to reduce flash usage.

Note: For more information about the EmberZNet token system, refer to both the token.h file and to the *EmberZNet Application Developer's Reference Manual* (120-3021-000).

Simulated EEPROM

Because the EM250 does not contain an internal EEPROM, a Simulated EEPROM (also referred to as sim-eeprom and SimEE) has been implemented to use 8KB of upper flash memory for stack and application token storage. Because the flash cells are only qualified for up to 1,000 write cycles, the Simulated EEPROM implements a wear-leveling algorithm that effectively extends the number of write cycles for individual tokens into the tens to hundreds of thousands.

The Simulated EEPROM is designed to operate below the token module as transparently as possible. However, for some applications you may want to customize the behavior when a flash erase is required, because this process requires a 21 millisecond period during which interrupts cannot be serviced. You can use the `halSimEepromCallback()` function for this purpose—while the erase must be performed to maintain proper functioning, the application can schedule it to avoid interfering with any other critical timing events. This function has a default handler implemented in the ember-configuration.c file that will erase the flash immediately. Applications can override this behavior by defining EMBER_APPLICATION_HAS_CUSTOM_SIM_EEPROM_CALLBACK.

A status function is also available to provide basic statistics about the usage of the Simulated EEPROM. For an in-depth discussion of the Simulated EEPROM, its design, its usage, and other considerations, refer to the document *Using the Simulated EEPROM* (120-6003-000).

Peripheral access

The EmberZNet networking stack requires access to certain on-chip peripherals; additionally, applications may use other on-chip or on-board peripherals. The default HAL provides implementations for all required peripherals and also for some commonly used peripherals. **Ember recommends that developers implement additional peripheral control within the HAL framework to facilitate easy porting and upgrade of the stack in the future.**

Note: Peripheral control provided by the specific version of the EmberZNet stack can be found by referring to the *HAL API Reference* "Sample APIs for Peripheral Access." An individual *HAL API Reference* is available for each Ember platform.

System timer control

The EmberZNet stack uses the system timer to control low-resolution timing events on the order of seconds or milliseconds. High-resolution (microsecond-scale) timing is managed internally through interrupts. Ember encourages developers to use the system timer control or the event controls (see Chapter 2 on the EmberZNet API) whenever possible; this helps to avoid replicating functionality and using scarce flash space unnecessarily. For example, you can use the function `halCommonGetInt16uMillisecondTick()`to check a previously stored value against the current value and implement a millisecond-resolution delay.

Bootloading

Bootloading functionality is also abstracted in the HAL interface. Refer to the *HAL API Reference* for your Ember platform for specifics on the platform being used, as well as the *EmberZNet Application Developer's Reference Manual* (120-3021-000) for a detailed description on the use and implementation of the bootloaders.

HAL utilities

The HAL utilities include general-purpose APIs that may rely on hardware capabilities (for example, CRC calculation that may take advantage of hardware acceleration). Crash and watchdog diagnostics, random number generation, and CRC calculation are provided by default in the HAL utilities.

Debug channel

The EmberZNet HAL implements a debug channel for communication with InSight Desktop. The debug channel provides a two-way out-of-band mechanism for the EmberZNet stack and customer applications to send debugging statistics and information to InSight Desktop for large-scale analysis. It provides API traces, debugging printfs, assert and crash information, and Virtual UART support when used with a DEBUG build of the EmberZNet stack. The DEBUG stack is larger than the DEBUG_OFF stack due to the debug and trace code.

Note: There are three levels of builds provided: DEBUG provides full debug capabilities, NORMAL provides Virtual UART but not full debug capabilities, and DEBUG_OFF removes all debug and Virtual UART support.

On the AVR/EM2420 platform, the debug channel utilizes one of the UARTs on the AVR for its communication. With the EM250, the SIF interface on the InSight Port is used for the debug channel in addition to IDE level debugging.

Virtual UART

EmberZNet supports Virtual UART functionality with DEBUG and NORMAL builds. The Virtual UART allows normal serial APIs to still be used on the port being used by the debug channel for debug output. For the EM250, which only has a single physical UART numbered as port 1, the Virtual UART always occupies port 0.

Virtual UART support is automatically enabled on AVR/EM2420 debug builds when EMBER_SERIALn_DEBUG is defined along with EMBER_SERIALn_MODE and appropriate serial queue definitions for the same port. If you do not want Virtual UART support, only define EMBER_SERIALn_DEBUG. For the EM250, the EMBER_SERIALn_DEBUG definition is not required since the debug channel operates over the InSight Port.

When Virtual UART support is enabled on the AVR/EM2420, any serial output sent to the same port being used by the debug channel is encapsulated in the debug channel protocol prior to being sent out to the physical UART. With the EM250, serial output sent to port 0 is encapsulated in the debug channel protocol and sent via the InSight Port. The raw serial output will be displayed by InSight Desktop, and will also appear on port 4900 of the adapter. Similarly, data sent to port 4900 of the adapter will be encapsulated in the debug channel protocol and sent to the node. The raw input data can then also be read using the normal serial APIs.

The Virtual UART allows custom application debug interfaces and output printfs to remain the same on AVR/EM2420 both for early release builds and debug builds that support the full EmberZNet debug functionality. With the EM250, it provides an additional port for output with debug builds that would otherwise not be available.

The following behaviors for the Virtual UART differ from normal serial UART behavior:

- `emberSerialWaitSend()` does not wait for data to finish transmitting
- `emberSerialGaruanteedPrintf()` is not guaranteed
- `EMBER_SERIALn_BLOCKING` might not block

More serial output might be dropped than normal depending on how busy the processor is with other stack functions.

Packet Trace support

EmberZNet supports a PacketTrace interface for use with InSight Desktop. This capability allows InSight Desktop to see all packets that are received and transmitted by all nodes in a network with no intrusion on the operation of those nodes. The PacketTrace interface works with both the dev0455 and dev0222 Ember developer kit carrier boards running any application.

Custom EM250 boards must have an InSight Port to use Packet Trace functionality. See the hardware chapter of the *Application Developer's Reference Manual* (120-3021-000) for a description of the InSight Port.

Custom EM2420 boards must have a Packet Trace port as described in the hardware chapter of the *Application Developer's Reference Manual* (120-3021-000) for access to packet data on this hardware.

In addition to the proper hardware connections to use Packet Trace functionality, the `BOARD_HEADER` must define the PACKET_TRACE macro. You can use the settings in dev0455.h for EM250 or dev0222.h for AVR/EM2420 as a template. For the AVR/EM2420, you must also define the appropriate ports and pins being used.

The PacketTrace interface works with both debug and non-debug builds as this support is provided by the hardware.

Customizing the HAL

This section describes how an end user would adapt the Ember-supplied standard HAL to their specific hardware and application requirements.

Compile-time configuration

The following preprocessor definitions are used to configure the EmberZNet HAL. They are usually defined in the Project file, but depending on the compiler configuration they may be defined in any global preprocessor location.

Required definitions

The following preprocessor definitions must be defined:

- `PLATFORM_HEADER`: The location of the platform header file. For example, the EM250 uses `hal/micro/xap2b/em250`.
- `BOARD_HEADER`: The location of the board header file. For example, the EM250 developer board uses `hal/micro/xap2b/em250/board/dev0455.h`. Custom boards should change this value to the new file name.
- `PLATFORMNAME`, such as XAP2B or AVR_ATMEGA.
- `PLATFORMNAME_MICRONAME` (for example, `XAP2B_EM250` or `AVR_ATMEGA_128`).
- `PHY_PHYNAME` (for example, `PHY_EM250` or `PHY_EM2420`).
- `BOARD_BOARDNAME` (for example, `BOARD_DEV0455` or `BOARD_DEV0222`).
- `CONFIGURATION_HEADER`: Provides additional custom configuration options for `ember-configuration.c`.

Optional definitions

The following preprocessor definitions are optional:

- `APPLICATION_TOKEN_HEADER`: When using custom token definitions, this preprocessor constant is the location of the custom token definitions file.
- `DISABLE_WATCHDOG`: This preprocessor definition can completely disable the watchdog without editing code. Use this definition very sparingly and only in utility or test applications, because the watchdog is critical for robust applications.

- EMBER_SERIALn_MODE = EMBER_SERIAL_FIFO or EMBER_SERIAL_BUFFER (*n* is the appropriate UART port). Leave this undefined if this UART is not used by the serial driver. Note that the Buffer serial mode on the EM250 also enables DMA buffering functionality for the UART.
- EMBER_SERIALn_TX_QUEUE_SIZE = power of 2 <= 128 (*n* is the appropriate UART port). This must be defined if EMBER_SERIALn_MODE is defined for this UART port. In FIFO mode, the value of this definition specifies the queue size in bytes. In Buffer mode, the definition represents a queue size as a number of packet buffers, each of which is *PACKET_BUFFER_SIZE* bytes (32 bytes as of this writing).
- EMBER_SERIALn_RX_QUEUE_SIZE = power of 2 <= 128 (*n* is the appropriate UART port). Must be defined if EMBER_SERIAL*n*_MODE is defined for this UART port. This value is always quantified in bytes (even in Buffer mode).
- EMBER_SERIALn_BLOCKING (*n* is the appropriate UART port). This must be defined if this serial port uses blocking IO (note that Ember does not recommend this for most applications).
- EMBER_SERIALn_DEBUG: Used for serial-based debug channels. *n* should always be 0 for the EM2420 and EM250.

Custom PCBs

Custom board header file

Creating a custom board is most easily done by modifying a copy of an existing board header file to match the configuration of the custom board. The board header file includes definitions for all the pinouts of external peripherals used by the HAL as well as macros to initialize and power up and down these peripherals. The board header is identified via the BOARD_HEADER preprocessor definition specified at compile time.

You can use the EM250 developer kit carrier board header file dev0455.h as a template when creating a new board header. Modify the port names and pin numbers used for peripheral connections as appropriate for the custom board hardware. These definitions can usually be easily determined by referring to the board's schematic.

Change the preprocessor definition BOARD_HEADER for this project to refer to the new filename.

In addition to the pinout modification, functional macros are defined within the board header file and are used to initialize, power up, and power down any board-specific peripherals. The macros are:

- halInternalInitBoard
- halInternalPowerDownBoard
- halInternalPowerUpBoard

Within each macro, you can call the appropriate helper halInternal APIs or, if the functionality is simple enough, insert the code directly.

Certain modifications might require you to change additional source files in addition to the board header. Situations that might require this include:

- Using different external interrupts or interrupt vectors
- Functionality that spans multiple physical IO ports
- Changing the core peripheral used for the functionality (for example, using a different timer or SPI peripheral)

In these cases, refer to the section "Modifying the default implementation."

Modifying the default implementation

The functionality of the EmberZNet HAL is grouped into source modules with similar functionality. These modules—the source files—can be easily replaced individually, allowing for custom implementations of their functionality. Table 1 summarizes the HAL source modules.

Table 1. EmberZNet 3.1 HAL Source Modules

Source Module	Description
adc	Sample functionality for accessing analog-to-digital converters built into the AVR and EM250 (refer to the document *Using the EM250 ADC* (120-5042-000) for additional information)
bootloader-interface-app	APIs for using the application bootloader
bootloader-interface-standalone	APIs for using the standalone bootloader
button	Sample functionality that can be used to access the buttons built into the developer kit carrier boards
buzzer	Sample functionality that can play notes and short tunes on the buzzer built into the developer kit carrier boards
crc	APIs that can be used to calculate a standard 16-bit CRC or a 16-bit CCITT CRC as used by 802.15.4
diagnostic	Sample functionality that can be used to help diagnose unknown watchdog resets and other unexpected behavior
em2420	APIs that are used by the EmberZNet Stack to access the EM2420 radio
flash	Internal HAL utilities used to read, erase, and write Flash in the EM250
led	Sample functionality that can be used to manipulate LEDs
mem-util	Common memory manipulation APIs such as memcopy
micro	Core HAL functionality to initialize, put to sleep, shutdown, and reboot the microcontroller and any associated peripherals
random	APIs that implement a simple pseudo-random number generator that is seeded with a true-random number when the EmberZNet Stack is initialized
rc-calibrate	Sample functionality that can be used to calibrate the built-in RC oscillators of the AVR
sim-eeprom	Simulated EEPROM system for storage of tokens in the EM250
spi	APIs that are used to access the SPI peripherals
symbol-timer	APIs that implement the highly accurate symbol timer required by the EmberZNet Stack
system-timer	APIs that implement the basic millisecond time base used by the EmberZNet Stack
token	APIs to access and manipulate persistent data used by the EmberZNet Stack and many applications
uart	Low-level sample APIs used by the serial utility APIs to provide serial input and output

Before modifying these peripherals, be sure you are familiar with the naming conventions and the hardware datasheet, and take care to adhere to the original contract of the function being replaced. **Ember recommends that you contact Ember Support before beginning any customization of these functions** to determine the simplest way to make the required changes.

4 Home Automation Application

This chapter describes the operation of the Home Automation sample application. This application provides a reference implementation of the ZigBee Home Automation Application Profile operating on the ZigBee PRO stack. You can use this application as the basis for developing Home Automation compliant applications.

The Home Automation sample application is a template application designed to provide ZigBee-compliant Home Automation sample code for developing products. This reference implementation was created using a configuration tool named the AppBuilder. See the InSight Desktop online help for a description of how to use the AppBuilder for generating a sample application.

Note that Ember intends to have this reference implementation ZigBee certified as a Home Automation application. At this time, that has not been completed. However, customer implementations will separately need to pass ZigBee certification testing if desired by implementer. This code is provided to jump start the development process.

Expected Operation and Data Flow

Sample code generated from the AppBuilder, such as this reference application, is designed to support any of the ZigBee clusters defined by the Home Automation Profile. Not all clusters are currently implemented but more will be added as needed by customers. In addition, this application will continue to grow as additional clusters are added to the ZigBee cluster library by other application profiles. A list of cluster commands (Table 3) and a list of attributes (Table 4) Ember supports are included later in this chapter.

The Home Automation network is a simple ZigBee network using network-level security and the optional use of link keys. Network keys are sent in the clear to joining devices. Permit joining is turned on in a network based on button pushes or other user interaction. When permit joining is on, any device that tries to join is accepted. Permit joining may only be turned on for limited duration.

Table 2 lists the typical Home Automation devices supported.

Table 2. Supported Devices

Device	Implements the Clusters for
Thermostat	HA device 0x301
Dimmable light	HA device 0x101
Dimmer switch	HA device 0x104
Mains powered outlet	HA device 0x9
Heating/cooling unit	HA device 0x300
Remote control	HA device 0x6
Temperature sensor	HA device 0x302

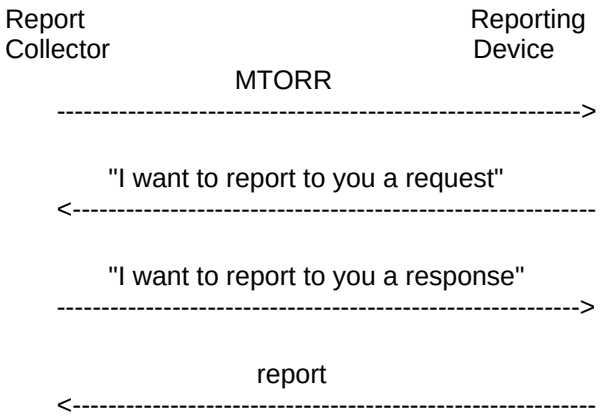
Note: You can build other Home Automation device types by adding the proper `#defines` to include the correct clusters.

For particular clusters, there are attributes that can be reported. To show a typical application, the AppBuilder allows turning on and off reporting of attributes. Ember recommends that you leave attribute reporting on to show some typical data flow.

The out-of-box reporting configuration is done to a central aggregation device. An aggregation device is used to minimize route discovery broadcasts and routing table entries.

The data flow for the reporting is shown below. The aggregation device is defined by setting an HA_CONCENTRATOR. This device will send regular many-to-one route requests (MTORR). Devices respond with a command indicating they will report attributes. The concentrator responds to this message to inform the device the reports are wanted. These reporting messages are the global reporting configuration messages with a manufacturer-specific bit set in the ZCL header and with a manufacturer's ID in the header.

Figure 1. Out-of-box reporting data flow



Other data flow can be established by discovering a cluster of interest on another device on the network (using the `zcl global discover [cluster] [attrID:2] [max # to report]` command) and then sending commands to devices that respond to the discovery.

Notes and limitations

Ember intends that the Home Automation application provide a base application framework and the over-the-air message formats for HA-compliant messages. You can modify the template code as it is all provided as source. Such modification could have an impact on ZigBee certification and should be done carefully.

This application uses `#defines` to enable pieces of the application code. See the app/ha/ha-config.h file for examples of the `#defines` you can use. A device is defined by the clusters it supports, so the base of the application stays the same and the clusters supported change. Supporting a client-side of a cluster enables the serial commands for that cluster. Supporting the server-side of a cluster defines the attributes in Table 4 and support for receiving messages generated by the client.

This application has not completed ZigBee certification testing and therefore could change as a result of this testing.

Customer usage of the template code requires integration of the specific hardware implementation and customization of the code for specific functions. For example, the sample temperature

implementation is based on using the ADC on the Ember development board. Your implementation should remove this and include your method for measuring temperature.

This application has been fully tested and validated within the Ember test network for typical expected network operations. If you encounter difficulties or identify problems, please send questions to support@ember.com. Some Home Automation networks use different configurations and network design methodology, and the ZigBee certification process allows this. As such, you should not feel that any Home Automation profile application must be based solely on this reference application. This is simply provided as one method for designing such an application.

Application Interface and User Operation

The HA application as written provides button interfaces for use on Ember development kit hardware and a serial interface for interaction with devices.

It uses the following buttons:

- BUTTON 0: if not joined: FORM (if the device is capable of forming)
- BUTTON 0: if joined: BIND (send ZDO end device bind request)
- BUTTON 1: if not joined: JOIN
- BUTTON 1: if joined: PERMIT JOINING

The serial port is set to 115200 baud on UART1 for all devices. The general serial commands supported are listed below.

To start the network, one device must be given a network form command. Once the network is formed, the network `pjoin` command is used to turn on permit joining, and other devices can then be given the network `join` command. Note that both the `form` and `join` commands require you to specify the channel, radio power, and panid in hex format. If default reporting is on, you can use InSight and view basic network operation once the network is formed and new devices join. You can then interact using serial commands to send and receive messages between devices to replicate expected operation of actual Home Automation devices.

The general serial commands on all devices are:

- help—shows what top-level commands are available
- version—gives the stack and application version
- info—gives information about the local node
- reset
- network form [channel] [power] [panid in hex]
- network join [channel] [power] [panid in hex]
- network leave
- network pjoin [time]
- option button0—simulate button0 press
- option button1—simulate button1 press
- send [id] [src endpoint] [dst endpoint]—send a message (need to build first using zcl scommand)
- print attr—print the ZCL attribute table
- print identify—print the identify cluster state
- print groups—print the groups table
- print scenes —print the scenes table
- print c (ias-ace info)—print the ias-ace info

- print report—print the reporting table
- write [cluster] [attrID] [dataLen] [data]
- zcl raw [cluster] [len] [data 0-8] [data 9-16] [data 17-24] [data 25-32]
- stats—if HA_ENABLE_STATS is defined, then this command shows the number of received and transmitted messages

There are additional global serial commands used for clusters. Because these commands construct a payload, you must issue a `send` call to send the message:

- zcl global read [cluster] [attrID:2]
- zcl global write [cluster] [attrID:2] [type] [data]
- zcl global uwrite [cluster] [attrID:2] [type] [data]
- zcl global nwrite [cluster] [attrID:2] [type] [data]
- zcl global discover [cluster] [attrID:2] [max # to report]
- zcl global report-read [cluster] [attrID:2] [direction:1]
- zcl global send-me-a-report [cluster] [attrID:2] [type:1] [min:2] [max:2] [chg:1-4]
- zcl global expect-report-from-me [cluster] [attrID:2] [timeout:2]

There are also cluster specific serial commands that are available on a per cluster basis. The application must have defined the client side of the cluster to have access to all these commands. Because these commands construct a payload, you must issue a `send` call to send the message:

- zcl basic rtfd
- zcl identify id [identify time:2]
- zcl identify query
- zcl groups add [grp ID:2] [name:16]
- zcl groups view [grp ID:2]
- zcl groups get [count:1] [[groupID:2] * count]
- zcl groups remove [grp ID:2]
- zcl groups rmall
- zcl groups ad-if-id [grp ID:2] [name:16]
- zcl scenes add [groupId:2] [sceneId:1] [trans time:1] [name:n]
- zcl scenes view [groupId:2] [sceneId:1]
- zcl scenes remove [groupId:2] [sceneId:1]
- zcl scenes rmall [groupId:2]
- zcl scenes store [groupId:2] [sceneId:1]
- zcl scenes recall [groupId:2] [sceneId:1]
- zcl scenes get-membership [groupId:2]
- zcl scenes set [on/off:1 boolean] [level:1 int]
- zcl on-off off
- zcl on-off on
- zcl on-off toggle
- zcl level-control mv-to-level [level:1] [trans time:2]
- zcl level-control move [mode:1] [rate:2]
- zcl level-control step [step:1] [step size:1] [trans time:2]
- zcl level-control stop

- zcl level-control o-mv-to-level [level:1] [trans time:2]
- zcl level-control o-move [mode:1] [rate:2]
- zcl level-control o-step [step:1] [step size:1] [trans time:2]
- zcl level-control o-stop
- zcl tstat set [mode:1 int] [amount:1 int]
- zcl ias-zone enroll [zone type: 2 int] [manuf code: 2 int]
- zcl ias-zone sc [zone status: 2 int] [ext status: 1 int]
- zcl ias-ace arm [mode: 1 int]
- zcl ias-ace bypass [numZones: 1 int] [zone: * int]
- zcl ias-ace emergency
- zcl ias-ace fire
- zcl ias-ace panic
- zcl ias-ace getzm
- zcl ias-ace getzi

Simple Example

1. Use the `info` command to determine the information about the device, including the EUI64, channel, PAN ID, short ID, extended PAN ID, server clusters supported, and client clusters supported.
2. To start a network, use this command:

   network form <channel in decimal> <power in decimal> <shortPan in hex

   For instance:

   network form 11 1 00aa

   If the `form` command does not work, check the device type that was set for the application. Only devices that show up as type "coordinator" can issue a `network form` call.
3. To join a network, make sure permit joining is on by doing:

   network pjoin <duration>

   For instance, to permit a join for 60 seconds:

   network pjoin 60
4. To join a device to a network, use:

   network join <channel in decimal> <power in decimal> <shortPan in hex

   Make sure the parameters match the ones used when forming.
5. Once devices are joined, start sending ZigBee commands. You do this by building a command and then sending it. The command `info` shows what server and client clusters are supported. The command:

   print a

   prints the attribute table that shows the cluster and attribute information. This is useful when sending read attributes commands. To create a read attributes command use:

   zcl global read <cluster in decimal> <attribute in hex>

   For instance, to create a read attributes for basic cluster (0), attribute ZCL version (0000), use:

zcl global read 0 0000

To send this command use:

send <address>

For instance, to send to the coordinator (who is address 0000) use:

send 0000

To determine the short address of a device to send to, use the "info" command and look for nodeID.

For instance, if the command `print a` shows this:

```
idx clus / attr /type(len)/ rw    / data
00: 000A / 0000 / 23 (04) / WRITE / 00 01 50 05 (time)
```

then reading that attribute on a device with shortID 0x5AAF is done as follows:

zcl global read 10 0000

send 5AAF

The result will be as follows:

RX len 0B, clus 0x000A (time) FC 00 seq 00 cmd 01 payload[00 00 00 23 B8 50 01 00]

The payload is attribute ID (0x0000), status (0x00), type (0x23), and value (0x00 01 50 B8)

Supported HA Commands

Table 3 lists the supported cluster commands for the Home Automation sample application.

Table 3. List of Home Automation Cluster Commands

Cluster domain	Cluster	Sent by	Cluster command	M/O	Payload
Global	Any (see Note 1)	either	read attributes (0x00)	M	([attrID 2] * n)
Global	Any (see Note 1)	either	read attr resp (0x01)	M	([attr ID 2] [status 1] [data type 0/1] [data n]) * n
Global	Any (see Note 1)	either	write attributes (0x02)	M	([attrID 2] [type 1] [data n]) * n
Global	Any (see Note 1)	either	write attr undivided (0x03)	M	([attrID 2] [type 1] [data n]) * n
Global	Any (see Note 1)	either	write attr resp (0x04)	M	([status 1] [attr ID 2]) * n
Global	Any (see Note 1)	either	write attr no resp (0x05)	M	([attrID 2] [type 1] [data n]) * n
Global	Any (see Note 1)	cli	config reporting (0x06)	M	([attrID 2] [type 1] [min 2] [max 2] [rpt chg 0-n]) * n "send me a report"
Global	Any (see Note 1)	svr	config reporting (0x06)	M	([attrID 2] [rpt chg 0-n] [timeout 2]) * n "expect this report from me"
Global	Any (see Note 1)	either	config reporting resp (0x07)	M	([status 1] [dir 1] [attrID 2]) * n
Global	Any (see Note 1)	either	read reporting config (0x08)	M	([dir 1] [attrID 2]) * n
Global	Any (see Note 1)	either	read reporting config resp (0x09)	M	([dir 1] [attrID 2] [type 1] [min 2] [max 2] [rptchg 0/n] [timeout 2]) * n
Global	Any (see Note 1)	either	report attributes (0x0a)	M	([attrID 2] [type 1] [data n]) * n
Global	Any (see Note 1)	either	default response (0x0b)	M	[cmd 1] [status 1]
Global	Any (see Note 1)	either	discover attributes (0x0c)	M	[start attrID 2] [max num 1]
Global	Any (see Note 1)	either	discover attributes resp (0x0d)	M	[done 1] ([attrID 2] [type 1]) * n
General	Basic (0x0000)	cli	restore to factory defaults (0x00)	O	<none>
General	Identify (0x0003)	cli	identify (0x00)	M	[identify_time 2]
General	Identify (0x0003)	cli	identify query (0x01)	M	<none>
General	Identify (0x0003)	svr	identify query response (0x00)	M	[timeout 2]
General	Groups (0x0004)	cli	add group (0x00)	M	[grpID 2] [name n]
General	Groups (0x0004)	cli	view group (0x01)	M	[grpID 2]
General	Groups (0x0004)	cli	get group membership (0x02)	M	[count 1] ([grpID 2] * n)
General	Groups (0x0004)	cli	remove group (0x03)	M	[grpID 2]
General	Groups (0x0004)	cli	remove all groups (0x04)	M	<none>
General	Groups (0x0004)	cli	add group if identifying (0x05)	M	[grpID 2] [name n]

Cluster domain	Cluster	Sent by	Cluster command	M/O	Payload
General	Groups (0x0004)	svr	add group response (0x00)	M	[status 1] [grpID 2]
General	Groups (0x0004)	svr	view group response (0x01)	M	[status 1] [grpID 2] [name n]
General	Groups (0x0004)	svr	get group membership resp (0x02)	M	[capacity 1] [count 1] ([grpID 2] * n)
General	Groups (0x0004)	svr	remove group response (0x03)	M	[status 1] [grpID 2]
General	Scenes (0x0005)	cli	add scene (0x00)	M	[grpID 2] [sceneID 1] [trans time 2] [name n] [ext n]
General	Scenes (0x0005)	cli	view scene (0x01)	M	[grpID 2] [sceneID 1]
General	Scenes (0x0005)	cli	remove scene (0x02)	M	[grpID 2] [sceneID 1]
General	Scenes (0x0005)	cli	remove all scenes (0x03)	M	[grpID 2]
General	Scenes (0x0005)	cli	store scene (0x04)	M	[grpID 2] [sceneID 1]
General	Scenes (0x0005)	cli	recall scene (0x05)	M	[grpID 2] [sceneID 1]
General	Scenes (0x0005)	cli	get scene membership ((0x06)	M	[grpID 2]
General	Scenes (0x0005)	svr	add scene response (0x00)	M	[status 1] [grpID 2] [sceneID 1]
General	Scenes (0x0005)	svr	view scene response (0x01)	M	[status 1] [grpID 2] [sceneID 1] [trans time 2] [name n] [ext n]
General	Scenes (0x0005)	svr	remove scene response (0x02)	M	[status 1] [grpID 2] [sceneID 1]
General	Scenes (0x0005)	svr	remove all scenes resp (0x03)	M	[status 1] [grpID 2]
General	Scenes (0x0005)	svr	store scene response (0x04)	M	[status 1] [grpID 2] [sceneID 1]
General	Scenes (0x0005)	svr	get scene mship resp ((0x06)	M	[status 1] [capacity 1] [grpID 2] [scene count 1] [scene list n]
General	On/Off (0x0006)	cli	off (0x00)	M	<none>
General	On/Off (0x0006)	cli	on (0x01)	M	<none>
General	On/Off (0x0006)	cli	toggle (0x02)	M	<none>
General	Level-Control (0x0008)	cli	move to level (0x00)	M	[level 1] [trans time 2]
General	Level-Control (0x0008)	cli	move (0x01)	M	[move mode 1] [rate 2]
General	Level-Control (0x0008)	cli	step (0x02)	M	[step 1] [step size 1] [trans time 2]
General	Level-Control (0x0008)	cli	stop (0x03)	M	<none>
General	Level-Control (0x0008)	cli	move to level w/on/off (0x04)	M	[level 1] [trans time 2]
General	Level-Control (0x0008)	cli	move w/on/off (0x05)	M	[move mode 1] [rate 2]
General	Level-Control (0x0008)	cli	step w/on/off (0x06)	M	[step 1] [step size 1] [trans time 2]
General	Level-Control (0x0008)	cli	stop w/on/off (0x07)	M	<none>
General	alarm (0x0009)	cli	reset alarm (0x00)	M	[alarm code 1] [clusterId 2]
General	alarm (0x0009)	cli	reset all alarms (0x01)	M	<none>

Cluster domain	Cluster	Sent by	Cluster command	M/O	Payload
General	alarm (0x0009)	cli	get alarm (0x02)	O	<none>
General	alarm (0x0009)	cli	reset alarm log (0x03)	O	<none>
General	alarm (0x0009)	svr	alarm (0x00)	M	[alarm code 1] [clusterId 2]
General	alarm (0x0009)	svr	get alarm response (0x01)	O	[status 1] [alarm code 1] [clusterId 2] [time stamp 4]
HVAC	Thermostat (0x0201)	cli	setpoint raise/lower (0x00)	M	[mode 1] [amount 1]
Lighting	Color Control (0x0300)	cli	move to hue (0x00)	M	[hue 1] [dir 1] [ttime 2]
Lighting	Color Control (0x0300)	cli	move hue (0x01)	M	[move mode 1] [rate 1]
Lighting	Color Control (0x0300)	cli	step hue (0x02)	M	[step mode 1] [ttime 1]
Lighting	Color Control (0x0300)	cli	move to saturation (0x03)	M	[saturation 1] [ttime 2]
Lighting	Color Control (0x0300)	cli	move saturation (0x04)	M	[move mode 1] [rate 1]
Lighting	Color Control (0x0300)	cli	step saturation (0x05)	M	[step mode 1] [ttime 1]
Lighting	Color Control (0x0300)	cli	move to hue and saturation (0x06)	M	[hue 1] [saturation 1] [ttime 2]
Security & Safety	IAS Zone(0x0500)	cli	zone enroll response (0x00)	M	[enroll status 1] [zone ID 1]
Security & Safety	IAS Zone(0x0500)	svr	zone status change notification (0x00)	M	[zone status 2] [extended status 1]
Security & Safety	IAS Zone(0x0500)	svr	zone enroll request (0x01)	M	[zone type 2] [mfg code 2]
Security & Safety	IAS ACE (0x0501)	cli	arm (0x00)	M	[arm mode 1]
Security & Safety	IAS ACE (0x0501)	cli	bypass (0x01)	M	[num zones 1] ([zoneId 2] * N)
Security & Safety	IAS ACE (0x0501)	cli	emergency (0x02)	M	<none>
Security & Safety	IAS ACE (0x0501)	cli	fire (0x03)	M	<none>
Security & Safety	IAS ACE (0x0501)	cli	panic (0x04)	M	<none>
Security & Safety	IAS ACE (0x0501)	cli	get zone id map (0x05)	M	<none>
Security & Safety	IAS ACE (0x0501)	cli	get zone information (0x06)	M	[zone id 1]
Security &	IAS ACE (0x0501)	svr	arm response (0x00)	M	[arm notification 1]

Cluster domain	Cluster	Sent by	Cluster command	M/O	Payload
Safety					
Security & Safety	IAS ACE (0x0501)	svr	get zone id map response (0x01)	M	[zone id map 32]
Security & Safety	IAS ACE (0x0501)	svr	get zone info response (0x02)	M	[zone id 1] [zone type 1] [IEEE 8]
Security & Safety	IAS WD (0x0502)	cli	start warning (0x00)	M	[warn 4b strobe 2b res 2b] [warn dur 2]
Security & Safety	IAS WD (0x0502)	cli	squawk (0x01)	M	[squawk mode 4b strobe 1b res 1b squawk level 2b]
	Note 1 = cluster only used to determine which attribute				

Supported HA Attributes

Table 4 lists the supported attributes for the Home Automation sample application.

Table 4. List of Home Automation Attributes

Cluster domain	Cluster ID	Cluster name	Attribute ID	Attribute	Data type	Data type ID	M / O	Access	Range	Default	Notes
General	0x0000	Basic	0x0000	ZCL version	int8u	0x20	M	R	any	0	
General	0x0000	Basic	0x0001	application version	int8u	0x20	O	R	any	0	
General	0x0000	Basic	0x0002	stack version	int8u	0x20	O	R	any	0	
General	0x0000	Basic	0x0007	power source	enum8	0x30	M	R	any	0	
General	0x0000	Basic	0x0012	device enabled	boolean	0x10	M	W/R	0-1	1	
General	0x0001	Power Configuration	0x0000	mains voltage	int16u	0x21	O	R	any	n/a	
General	0x0001	Power Configuration	0x0001	mains frequency	int8u	0x20	O	R	any	n/a	
General	0x0001	Power Configuration	0x0010	mains alarm mask	bitmap8	0x18	O	W/R	0-3	0	
General	0x0001	Power Configuration	0x0011	mains voltage min thresh	int16u	0x21	O	W/R	any	0	
General	0x0001	Power Configuration	0x0012	mains voltage max thresh	int16u	0x21	O	W/R	any	0xffff	
General	0x0001	Power Configuration	0x0013	mains voltage dwell trip	int16u	0x21	O	W/R	any	0	
General	0x0001	Power Configuration	0x0020	battery voltage	int8u	0x20	O	R	any	n/a	
General	0x0001	Power Configuration	0x0030	battery manufacturer	char string	0x42	O	W/R	0-16 b	empty	
General	0x0001	Power Configuration	0x0031	battery size	enum8	0x30	O	W/R	any	0xff	
General	0x0001	Power Configuration	0x0032	battery AHr rating	int16u	0x21	O	W/R	any	n/a	
General	0x0001	Power Configuration	0x0033	battery quantity	int8u	0x20	O	W/R	any	n/a	
General	0x0001	Power Configuration	0x0034	battery rated voltage	int8u	0x20	O	W/R	any	n/a	
General	0x0001	Power Configuration	0x0035	battery alarm mask	bitmap8	0x18	O	W/R	0-1	0	

Cluster domain	Cluster ID	Cluster name	Attribute ID	Attribute	Data type	Data type ID	M / O	Access	Range	Default	Notes
General	0x0001	Power Configuration	0x0036	battery voltage min thresh	int8u	0x20	O	W/R	any	0	
General	0x0002	Device Temperature Configuration	0x0000	current temperature	int16s	0x29	M	R	. -200 to +200	n/a	
General	0x0002	Device Temperature Configuration	0x0001	min temp experienced	int16s	0x29	O	R	. -200 to +200	n/a	
General	0x0002	Device Temperature Configuration	0x0002	max temp experienced	int16s	0x29	O	R	. -200 to +200	n/a	
General	0x0002	Device Temperature Configuration	0x0003	over temp total dwell	int16s	0x29	O	R	any	0	
General	0x0002	Device Temperature Configuration	0x0010	device temp alarm mask	bitmap8	0x18	O	W/R	any	0	
General	0x0002	Device Temperature Configuration	0x0011	low temp threshold	int16s	0x29	O	W/R	. -200 to +200	n/a	
General	0x0002	Device Temperature Configuration	0x0012	high temp threshold	int16s	0x29	O	W/R	. -200 to +200	n/a	
General	0x0002	Device Temperature Configuration	0x0013	low temp dwell trip point	int24u	0x22	O	W/R	any	n/a	
General	0x0002	Device Temperature Configuration	0x0014	high temp dwell trip point	int24u	0x22	O	W/R	any	n/a	
General	0x0003	Identify	0x0000	identify time	int16u	0x21	M	W/R	any	0	
General	0x0004	Groups	0x0000	name support	bitmap8	0x18	M	R	0 or 128	n/a	128 = supported; groups names 0-16 bytes
General	0x0005	Scenes	0x0000	scene count	int8u	0x20	M	R	any	0	
General	0x0005	Scenes	0x0001	current scene	int8u	0x20	M	R	any	0	
General	0x0005	Scenes	0x0002	current group	int16u	0x21	M	R	0 to 0xfff7	0	
General	0x0005	Scenes	0x0003	scene valid	boolean	0x10	M	R	0 or 1	0	
General	0x0005	Scenes	0x0004	name support	bitmap8	0x18	M	R	any	n/a	

Cluster domain	Cluster ID	Cluster name	Attribute ID	Attribute	Data type	Data type ID	M / O	Access	Range	Default	Notes
General	0x0005	Scenes	0x0005	last configured by	IEEE addr	0xF0	O	R	any	n/a	
General	0x0006	On/Off	0x0000	on / off	boolean	0x10	M	R	0 or 1	0	
General	0x0007	On/Off Switch Configuration	0x0000	switch type	enum8	0x30	M	R	0 or 1	n/a	
General	0x0007	On/Off Switch Configuration	0x0010	switch actions	enum8	0x30	M	W/R	0 to 2	0	
General	0x0008	Level Control	0x0000	current level	int8u	0x20	M	R	any	0	
General	0x0008	Level Control	0x0001	remaining time	int16u	0x21	O	R	any	0	
General	0x0008	Level Control	0x0010	on off transition time	int16u	0x21	O	W/R	any	0	
General	0x0008	Level Control	0x0011	on level	int8u	0x20	O	W/R	0 to 0xfe	0xfe	
General	0x0009	Alarm	0x0000	alarm count	int16u	0x21	O	R	any	0	
General	0x000a	Time	0x0000	time	int32u	0x23	M	W/R	any	n/a	
General	0x000a	Time	0x0001	time status	bitmap8	0x18	M	W/R	0 to 3	0	
Closures	0x0100	Shade Configuration	0x0000	physical closed limit	int16u	0x21	O	R	1 to 0xfffe	n/a	
Closures	0x0100	Shade Configuration	0x0001	motor step size	int8u	0x20	O	R	0 to 0xfe	n/a	
Closures	0x0100	Shade Configuration	0x0002	status	bitmap8	0x18	M	W/R	0 to 15	0	
Closures	0x0100	Shade Configuration	0x0010	closed limit	int16u	0x21	M	W/R	1 to 0xfffe	1	
Closures	0x0100	Shade Configuration	0x0011	mode	enum8	0x30	M	W/R	0 to 0xfe	0	
HVAC	0x0200	Pump Configuration and Control	0x0000	max pressure	int16s	0x29	M	R	0x8001 to 0x7FFF	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0001	max speed	int16u	0x21	M	R	0 to 0xFFFE	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0002	max flow	int16u	0x21	M	R	0 to 0xFFFE	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0003	min const pressure	int16s	0x29	O	R	0x8001 to 0x7FFF	n/a	

Cluster domain	Cluster ID	Cluster name	Attribute ID	Attribute	Data type	Data type ID	M / O	Access	Range	Default	Notes
HVAC	0x0200	Pump Configuration and Control	0x0004	max const pressure	int16s	0x29	O	R	0x8001 to 0x7FFF	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0005	min comp pressure	int16s	0x29	O	R	0x8001 to 0x7FFF	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0006	max comp pressure	int16s	0x29	O	R	0x8001 to 0x7FFF	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0007	min const speed	int16u	0x21	O	R	0 to 0xFFFE	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0008	max const speed	int16u	0x21	O	R	0 to 0xFFFE	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0009	min const flow	int16u	0x21	O	R	0 to 0xFFFE	n/a	
HVAC	0x0200	Pump Configuration and Control	0x000a	max const flow	int16u	0x21	O	R	0 to 0xFFFE	n/a	
HVAC	0x0200	Pump Configuration and Control	0x000b	min const temp	int16s	0x29	O	R	0x954D to 0x7FFF	n/a	
HVAC	0x0200	Pump Configuration and Control	0x000c	max const temp	int16s	0x29	O	R	0x954D to 0x7FFF	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0010	pump status	bitmap16	0x19	O	R	0 to 0x00FF	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0011	effective operation mode	enum8	0x30	M	R	0 to 0xFE	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0012	effective control mode	enum8	0x30	M	R	0 to 0xFE	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0013	capacity	int16s	0x29	M	R	0 to 0x7FFF	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0014	speed	int16u	0x21	O	R	0 to 0xFFFE	n/a	
HVAC	0x0200	Pump Configuration and Control	0x0015	lifetime running hours	int24u	0x22	O	W/R	0 to 0xFFFFFE	0	
HVAC	0x0200	Pump Configuration	0x0016	power	int24u	0x22	O	W/R	0 to 0xFFFFFE	n/a	

Cluster domain	Cluster ID	Cluster name	Attribute ID	Attribute	Data type	Data type ID	M / O	Access	Range	Default	Notes
		and Control									
HVAC	0x0200	Pump Configuration and Control	0x0017	lifetime energy consumed	int32u	0x23	O	R	0 to 0xFFFFFFFE	0	
HVAC	0x0200	Pump Configuration and Control	0x0020	operation mode	enum8	0x30	M	W/R	0 to 0xFE	0	
HVAC	0x0200	Pump Configuration and Control	0x0021	control mode	enum8	0x30	O	W/R	0 to 0xFE	0	
HVAC	0x0200	Pump Configuration and Control	0x0022	alarm mask	bitmap16	0x19	O	R	0 to 0x3FFF	n/a	
HVAC	0x0201	Thermostat	0x0000	local temperature	int16s	0x29	M	R	0x954D to 0x7FFF	n/a	
HVAC	0x0201	Thermostat	0x0011	occ cooling set point	int16s	0x29	M	W/R	min cool set limit - max cool set limit	0x0A28	26 degrees C
HVAC	0x0201	Thermostat	0x0012	occ heating setpoint	int16s	0x29	M	W/R	min heat set limit - max heat set limit	0x07D0	20 degrees C
HVAC	0x0201	Thermostat	0x0015	minheat	int16s	0x29	O	W/R	0x954D to 0x7FFF	0x02BC	7 degrees C
HVAC	0x0201	Thermostat	0x0016	maxheat	int16s	0x29	O	W/R	0x954D to 0x7FFF	0x0BB8	30 degrees C
HVAC	0x0201	Thermostat	0x0017	mincool	int16s	0x29	O	W/R	0x954D to 0x7FFF	0x02BC	7 degrees C
HVAC	0x0201	Thermostat	0x0018	maxcool	int16s	0x29	O	W/R	0x954D to 0x7FFF	0x0BB8	30 degrees C
HVAC	0x0201	Thermostat	0x0019	deadband	int8s	0x28	O	W/R	0x0a to 0x19	0x19	2.5 degrees C
HVAC	0x0201	Thermostat	0x001a	remotesensing	bitmap8	0x18	O	W/R	0 to 7	0	
HVAC	0x0201	Thermostat	0x001b	control seq	enum8	0x30	M	W/R	0 to 5	4	
HVAC	0x0201	Thermostat	0x001c	system mode	enum8	0x30	M	W/R	0 to 2	2	
HVAC	0x0202	Fan Control	0x0000	fan mode	enum8	0x30	M	W/R	0 to 6	5	

Cluster domain	Cluster ID	Cluster name	Attribute ID	Attribute	Data type	Data type ID	M / O	Access	Range	Default	Notes
HVAC	0x0202	Fan Control	0x0001	fan mode sequence	enum8	0x30	M	W/R	0 to 4	2	
HVAC	0x0203	Dehumidification Control	0x0000	relative humidity	int8u	0x20	O	R	0 to 0x64	n/a	
HVAC	0x0203	Dehumidification Control	0x0001	dehumidification cooling	int8u	0x20	M	R	0 to dehumidMaxCool	n/a	
HVAC	0x0203	Dehumidification Control	0x0010	RH dehumid setpoint	int8u	0x20	M	W/R	0x1E to 0x64	0x32	
HVAC	0x0203	Dehumidification Control	0x0011	relative humidity mode	enum8	0x30	O	W/R	0 to 1	0x00	
HVAC	0x0203	Dehumidification Control	0x0012	dehumid lockout	enum8	0x30	O	W/R	0 to 1	0x01	
HVAC	0x0203	Dehumidification Control	0x0013	dehumid hysteresis	int8u	0x20	M	W/R	0x02 to 0x14	0x02	
HVAC	0x0203	Dehumidification Control	0x0014	dehumid max cool	int8u	0x20	M	W/R	0x14 to 0x64	0x14	
HVAC	0x0203	Dehumidification Control	0x0015	relative humidity display	enum8	0x30	O	W/R	0 to 1	0x00	
HVAC	0x0204	Thermostat UI Config	0x0000	temperature display mode	enum8	0x30	M	W/R	0 to 1	0	
HVAC	0x0204	Thermostat UI Config	0x0001	keypad lockout	enum8	0x30	M	W/R	0 to 5	0	
Lighting	0x0300	Color Control	0x0000	current hue	int8u	0x20	M	R	0x00 to 0xFE	0	
Lighting	0x0300	Color Control	0x0001	current saturation	int8u	0x20	M	R	0x00 to 0xFE	0	
Lighting	0x0300	Color Control	0x0002	remaining time	int16u	0x21	O	R	0x0000 to 0xFFFE	0	
Measurement & Sensing	0x0400	Illuminance Measurement	0x0000	measured value	int16u	0x21	M	R	minMeas to maxMeas	0	
Measurement & Sensing	0x0400	Illuminance Measurement	0x0001	min measured value	int16u	0x21	M	R	0x0002 to 0xFFFD	n/a	
Measurement &	0x0400	Illuminance	0x0002	max measured	int16u	0x21	M	R	0x0001 to	n/a	

Cluster domain	Cluster ID	Cluster name	Attribute ID	Attribute	Data type	Data type ID	M / O	Access	Range	Default	Notes
Sensing		Measurement		value					0xFFFE		
Measurement & Sensing	0x0401	Illuminance Level Sensing	0x0000	level status	enum8	0x30	M	R	0x00 to 0xFE	n/a	
Measurement & Sensing	0x0401	Illuminance Level Sensing	0x0001	light sensor type	enum8	0x30	O	R	0x00 to 0xFE	n/a	
Measurement & Sensing	0x0401	Illuminance Level Sensing	0x0010	illuminance level target	int16u	0x21	M	W/R	0x0000 to 0xFFFE	n/a	
Measurement & Sensing	0x0402	Temperature Measurement	0x0000	measured value	int16s	0x29	M	R	min to max	0	
Measurement & Sensing	0x0402	Temperature Measurement	0x0001	min measured value	int16s	0x29	M	R	0x954d to 0x7fffe	n/a	
Measurement & Sensing	0x0402	Temperature Measurement	0x0002	max measured value	int16s	0x29	M	R	0x954e to 0x7ffff	n/a	
Measurement & Sensing	0x0403	Pressure Measurement	0x0000	measured value	int16s	0x29	M	R	min to max	0	
Measurement & Sensing	0x0403	Pressure Measurement	0x0001	min measured value	int16s	0x29	M	R	0x8001 to 0x7ffe	n/a	
Measurement & Sensing	0x0403	Pressure Measurement	0x0002	max measured value	int16s	0x29	M	R	0x8002 to 0x7ffff	n/a	
Measurement & Sensing	0x0404	Flow Measurement	0x0000	measured value	int16u	0x21	M	R	min to max	0	
Measurement & Sensing	0x0404	Flow Measurement	0x0001	min measured value	int16u	0x21	M	R	0x0000 to 0xfffd	n/a	
Measurement & Sensing	0x0404	Flow Measurement	0x0002	max measured value	int16u	0x21	M	R	0x0000 to 0xfffe	n/a	
Measurement & Sensing	0x0405	Relative Humidity Measurement	0x0000	measured value	int16u	0x21	M	R	min to max	0	
Measurement & Sensing	0x0405	Relative Humidity Measurement	0x0001	min measured value	int16u	0x21	M	R	0x0000 to 0x270f	n/a	
Measurement & Sensing	0x0405	Relative Humidity Measurement	0x0002	max measured value	int16u	0x21	M	R	0x0000 to 0x2710	n/a	

Cluster domain	Cluster ID	Cluster name	Attribute ID	Attribute	Data type	Data type ID	M / O	Access	Range	Default	Notes
Measurement & Sensing	0x0405	Relative Humidity Measurement	0x0003	tolerance	int16u	0x21	O	R	0x0000 to 0x0800	n/a	
Measurement & Sensing	0x0406	Occupancy Sensing	0x0000	occupancy	bitmap8	0x18	M	R	0 to 1	n/a	
Measurement & Sensing	0x0406	Occupancy Sensing	0x0001	occupancy sensor type	enum8	0x30	M	R	0 to 0xfe	n/a	
Measurement & Sensing	0x0406	Occupancy Sensing	0x0010	PIR occ to unocc delay	int8u	0x20	O	W/R	0 to 0xfe	0	
Measurement & Sensing	0x0406	Occupancy Sensing	0x0011	PIR unocc to occ delay	int8u	0x20	O	W/R	0 to 0xfe	0	
Measurement & Sensing	0x0406	Occupancy Sensing	0x0020	ultrasonic occ to unocc delay	int8u	0x20	O	W/R	0 to 0xfe	0	
Measurement & Sensing	0x0406	Occupancy Sensing	0x0021	ultrasonic unocc to occ delay	int8u	0x20	O	W/R	0 to 0xfe	0	
Security & Safety	0x0500	IAS Zone	0x0000	zone state	enum8	0x30	M	R	any	0	
Security & Safety	0x0500	IAS Zone	0x0001	zone type	enum8	0x30	M	R	any	n/a	
Security & Safety	0x0500	IAS Zone	0x0002	zone status	bitmap16	0x19	M	R	any	0	
Security & Safety	0x0500	IAS Zone	0x0010	IAS CIE address	IEEE addr	0xF0	M	W/R	any	n/a	
Security & Safety	0x0502	IAS WD	0x0001	max duration	int16u	0x21	M	W/R	0x0000 to 0xFFFE	240	

5 Sensor Sink Application

This chapter describes the operation and design of the Sensor Sink application. This application is designed as a simple aggregation system with a large number of sensor endpoints and a smaller number of data collectors (sinks). This application was developed by Ember prior to both the ZigBee Cluster Library and EmberZNet 3.1. It is not intended to be used as a model for a ZigBee-compliance application, but has been carried forward as a reference since many Ember customers are familiar with this application.

The application includes the following device types:

- **Sensor:** A device that takes data readings from some input source and passes these readings to a particular collection point. Many sensors report to a single sink.
- **Sink:** A device that serves as a collection point for 1 or more sensor devices. In this application, the sink is set up to be the ZigBee coordinator device, and it forms the network automatically on first startup and retains these settings across reboots.
- **Sleepy-sensor:** A sleeping (duty cycling) version of the Sensor Sink application. Battery powered devices must sleep in order to extend their battery life.
- **Mobile-sensor:** A sleeping and mobile version of the Sensor Sink application. This is for devices that are expected to move and therefore switch parents in the network. Sleepy devices may also switch parents, but it is assumed that mobile devices will be switching parents more often.

Expected Operation and Data Flow

This application assumes a sink is started up and forms the network. Other sinks may also then join the network and advertise their services. Sensors join the network and periodically send data into a sink.

All data flow is from sensors into the sink. The sink periodically updates the routes to the sensors. Sensors use aggregation routing to the sink, and the sink uses source routing back to the sensors. If a sensor moves, the source route is updated.

This application uses a "push" style of communication, where the sensor sends reports to the sink without needing to be asked for this data. This is more efficient than a "pull" model, where a device only transmits data when asked to do so by another device, because it cuts the amount of traffic in half, thereby reducing the number of collisions and routing burden in the network.

Notes and limitations

The following notes and limitations apply to this application.

1. This application defaults to using a fixed set of network parameters, with a default channel setting of 26. (Remove the `#define USE_HARDCODED_NETWORK_SETTINGS` line in the common.h header file to allow the application to dynamically select its network parameters.) Note that if you change any of these network parameters, the device must leave the network and join (or

form) again in order to apply the new settings. Simply rebooting the device will cause the node to resume its previous network settings (due to the use of the `emberNetworkInit()` API call).

2. Although the application portrays a single sink node that acts as a ZigBee Coordinator, you could easily adapt the application to allow a variant of the sink node that joins the network as an ordinary ZigBee Router so that multiple sinks can be supported.

3. Although the application is set up to have a sensor node participate in the network as a ZigBee Router, the application could support sensors as sleepy or mobile ZigBee end devices as well.

4. The application uses a constant called `MISS_PACKET_TOLERANCE` as a threshold for fault tolerance. On the sensor, this controls how many message timeouts can be permitted between the sensor and sink before the sensor decides to attach itself to a different sink. For a sink, this controls how many data reports can be missed from a sensor before the sink "forgets" about the sensor (stops maintaining a record of its attachment). Although bindings are kept on the sink node to track the attached sensors, these could easily be made temporary (used only for the `SINK_READY` message) to allow the Senor Sink application to support more nodes without enlarging the binding table, assuming that the application does not care which and how many sensors are attached to it. Alternatively, address table entries (which are not preserved across reboots of the node) could be used to track attached sensors.

5. This application is designed to scale to approximately 250 sensors and one sink. The applicability of this in large networks is dependent on the traffic rates expected from sensors to sink. Suitable jitter needs to be used when sending sensor data reports to ensure that traffic is spread out to avoid bottlenecks around the sink.

6. The sink advertisement is a broadcast. The behavior of such a broadcast in the network depends on the network topology and density. To minimize the network disruption and loss of bandwidth, ZigBee limits the number of broadcasts that can be active in a network to 10. As this network increases in size or density, the rate of the sink advertisement should be reduced in frequency.

Application Interface and User Operation

This application is designed for Ember development kits and has an interface that uses the breakout board buttons as well as the serial port. The button and serial interface used is described below. Note that the sink device automatically forms a network on startup, and therefore no user action is provided in this application. If a sink is reset, it will restart operation on the same network unless a Leave Network command has been given prior to the reset.

Sensor buttons used

- BUTTON 0: If the device is not joined to a network, pressing this causes the device to search for an available network and join it if possible. After it has successfully joined a network, pressing this button will cause the device to permit joining (accepting other devices into the network) for the next 60 seconds.
- BUTTON 1: Forces the device to reset. This can be useful if you have used the `!` command to leave the network and now want the device to attempt network participation again.

Note: If you have not left the network before resetting, the device will simply resume operation on the same network as before.

Sink buttons used

- BUTTON 0: After network formation, pressing this button allows other devices to join the network for the next 60 seconds.
- BUTTON 1: Forces the device to reset. This can be useful if you have used the `!` command to leave the network and now want the device to establish a new network.

Note: If you have not left the network before resetting, the device will simply resume operation on the same network as before.

Serial baud rates, ports used

- 115200 bps on UART1 for all devices

Serial commands supported

The following serial commands are used in this application. As noted, some of these commands are on sink and some are on sensor only.

- **f** - Forces the sink to advertise (sink only).
- **t** - Makes the node play a tune. Useful in identifying a node.
- **p** - Prints the node's binding table.
- **l** - Tells the node to send a multicast hello packet.
- **i** - Prints information about this node including channel, power, and app.
- **b** - Puts the node into the bootloader menu (as an example).
- **0** - Simulates button 0 press—turns permit join on for 60 seconds, allowing other nodes to join to this node.
- **1** - Simulates button 1 press—resets this node.
- **!** - Leaves the ZigBee network, so on node reset (button 1) the sink can form a new network.
- **x** - Prints token information (non-volatile settings stored in EEPROM).
- **c** - Prints child table (sensor and sink only).
- **j** - Prints status of Just In Time (JIT) message storage (JIT messages are used to communicate reliably with sleeping end devices).
- **a** - Used for debugging. Prints the status of `emberNetworkState`, `emberOkToNap`, `emberOkToHibernate`, and `emberCurrentStackTasks`. The application must be built with `DEBUG_NETWORK_STATE`.
- **?** - Prints the help menu.

6 Rangetest Application

This chapter describes the operation of the Rangetest application. This application is designed as a low-level test program for RF modules or other custom hardware. This program is generally used for setting up customer hardware, range or RF testing, and FCC or CE certification testing of devices. The program interface can be scripted for automated testing or run manually using interfaces such as Hyperterm.

The application does not include any ZigBee networking functionality or different device types. It is strictly a test application to provide access to the low-level hardware on EM250, EM260, and EM2420 devices. This application can be used between multiple devices, where one device is transmitting packets to other device(s) but these are not ZigBee-formatted messages.

For additional details about the functional testing process, including loading and using Rangetest on custom-built EM250 devices, refer to the application note *Bringing Up Custom Devices for the EM250 SoC Platform* (120-5031-000). If you are implementing a manufacturing test strategy, you should also consider reading the application note *Manufacturing Test Guidelines* (120-5016-000). For overall application testing and debugging strategies, refer to Chapter 10, Testing and Debug Strategies for ZigBee Application Development, in the *EmberZNet Application Developer's Reference Manual* (120-3021-000).

Expected Operation

This application does not perform any automatic operations without user interaction. All commands and functions are accessed through serial port commands. See the "Application Interface and User Operation" section for the commands and expected outcome. This is a test and development application used for bringing up hardware and device testing only. If you want to utilize similar low-level testing functionality outside of the pre-built Rangetest application, use the Manufacturing Library ("mfglib") API to accomplish this. The specifications for the manufacturing library APIs are detailed in the *Unified API Reference* documents as noted below for the EM250 and EM2420 respectively:

- *EmberZNet Unified API Reference – EM250* (120-3016-000)
- *EmberZNet Unified API Reference – EM2420* (120-3019-000)

For the EM260, the manufacturing library APIs are detailed in the *EZSP Reference Guide* (120-3009-000) in the section entitled "Mfglib frames."

Notes and limitations

The following notes and limitations apply to this application.

- This application is provided in pre-built form only.
- The packets transmitted by Rangetest are raw packets sent by the radio without the aid of a networking stack, so 802.15.4 and ZigBee conventions, such as CSMA-CA algorithms, random backoffs, transmission retries, and MAC and Network Layer packet headers do not apply in this

context. Such test conditions can produce very different results when comparing reliability of communication to a test case involving two (or more) devices running a full ZigBee networking stack. Therefore, Ember recommends that you perform interference testing, coexistence testing, and reliability testing with your own application (based on the networking stack) in addition to any similar testing done with Rangetest, so that your results are more representative of a true networking scenario.

- Packets transmitted by Rangetest are of a fixed size (the minimum size allowed by the radio) and are always transmitted at a fixed rate. (This rate may be different for different radios; see program output for details.)
- This application is not intended for use in operating networks. Use by nodes in a network will result in packet collisions and lost traffic. This application can be used by multiple nodes at the same time where one node sends packets and multiple other nodes receive the traffic.

Application Interface and User Operation

Buttons used

No buttons are used.

Serial baud rates, ports used

The EM250 version of Rangetest runs at 115200 bps. The active serial port is auto-detected; when either the UART or virtual UART first detects a Carrier Return (\r), Line Feed (\n), or asterisk (*) character, it becomes the port used. The UART or virtual UART remains the active port until the application is reset.

Serial commands supported

This application supports different commands depending on the hardware platform being used. Use the `help` or `?` command to get a listing of all supported commands on the current device's platform.

Table 5 lists the typical commands supported across the platforms.

Table 5. Serial Commands Supported

Command	Description
`?, help`	Prints the Help menu.
`bootload`	Launches the bootloader application.
`calchannel`	Uses `calchannel` *x* to switch to channel *x* and perform calibration (uses current channel if *x* is not specified). This command is not used on the EM2420.
`channel`	Sets the channel (11 by default).
`ledoff`	Use `ledoff` *x* to turn BOARDLEDx off.
`ledon`	Use `ledon` *x* to turn BOARDLEDx on.
`ledtest`	Cycles the 4 LEDs (at their default GPIOs on the EM250 Breakout Board) until told to stop.
`receive`	Puts the device into receive (RX) mode on the current channel. The following commands are valid while in receive mode: c—Clear statistics. q—Query statistics. e—Exit receive mode.
`shutdown`	Places the chip in the lowest power mode for deep sleep current measurements.
`sleep`	Sends the radio to sleep (x=1) or wakes it up (x=0).
`tokdump`	Dumps all known tokens and their values.
`tokread`	`tokread` <*key*> shows the contents of the token indexed by <*key*> as the stack will read it when run.

Command	Description
`tokscrub`	Erases and reinitializes simulated EEPROM, deleting all tokens stored there.
`tokwrite`	`tokwrite` <*key*> writes a new value to the token indexed by <*key*> and prompts for each byte of data. Manufacturing tokens cannot be written with this command.
`transmit`	Transmits the specified number of packets on the current channel (infinite if 0).
`txpow`	Sets power to specified dBm. For valid values, see *EM250 Radio Communication Module* technical specification (120-2001-000).
`txpowermode`	Uses `txpowermode` *x y* for *x*=0 or 1 and *y*=0 or 1 to engage Boost mode (*x*=1) for the chip or switch to using the external PA (`RF_TX_ALT_P/N`) signal path (*y*=1).
`txstream`	Performs a modulated carrier wave transmission on the current channel.
`txtone`	Performs an unmodulated carrier wave ("tone") transmission on the current channel.

A common use of the Rangetest application is to perform a simple send/receive test on a device to determine its range and generally test its radio functionality. The following procedure describes how to do this.

1. Connect two devices with the Rangetest application to a computer using a serial port. The devices must be in good operating order.
2. Connect to each device using a terminal emulator such as Hyperterm. Make sure that Rangetest is running on the new device (you should see the > prompt).
3. Set both devices to a channel by typing `channel` *X*, where *X* is the channel.
4. Optional: Set a power level on the test device by typing `txpow`. When prompted, specify the power level to use.
5. Optional: Engage Boost mode or the external PA signal path using the `txpowermode` command with the appropriate arguments. (See the commands in Table 5 for details about command usage.)
6. On the other device, type `receive`, which sets the device to receive and display statistics for each packet received. The statistics fields are as follows:

Field	Meaning
crf	Failed CRC on a packet. Updated stat only appears when the next good packet arrives.
err	Percentage of failed packets, calculated from sequence numbers non-sequentially received.
mis	Missed packets due to non-sequential sequence numbers.
ovf	Number of overflow errors. Stat is only updated when the next good packet is received.
pck	Total packets received correctly with CRC.
seq	Sequence number received in the packet.
spd	Failures detected at start-of-packet delimiters (SPD).
tot	Total packets sent (calculated from sequence numbers).

7. On the test device, type `transmit 64` to transmit 100 (64 hex) packets. (Note that `transmit 0` sends infinite packets.)
8. Type `e` to exit on the transmit mode to stop at any time.

Reverse this procedure to test receiving on the test device.

On the receive device, you can type `c` to clear the statistics, `q` to query statistics, and `e` to exit receive mode.

7 Training Application

The Ember Training application lets you interactively control an EmberZNet node via a self-documenting serial command line interface. Using any terminal program, you simply connect to the host's serial port and enter commands and parameters in ASCII text. The application prints command responses, incoming data traffic, and stack callbacks and notifications to the screen in an easy-to-read format. The *EZSP Reference Guide* (120-3009-000) contains detailed information about the EZSP command set. Ember highly recommends reading this document if you are using the Training application.

With the Training application, you can quickly and easily form networks; manage endpoints and bindings; send APS, multicast, and transport messages; create aggregation routes; and configure security; in short, this lets you explore all the powerful networking features of EmberZNet on actual hardware without writing a line of code. This accelerates your understanding of EmberZNet capabilities, networking concepts, and APIs before starting application development, which directly and dramatically reduces your debugging time and time to market.

Developers who like to dive in and get their hands dirty without reading the manual will love the Training application's built-in interactive Help menu. (Yes, we admit to knowing a few people like that.) Most command names and parameters correspond directly to EmberZNet functions, so you'll be learning the API as you go. All names, status values, and constants are displayed in their unabbreviated English form, so you won't have to flip through reference manuals to look up what status 0x66 or node type 2 means. And for those in a hurry, common commands have one-character shortcuts.

The richly formatted output provides complete visibility into application layer events such as network formation and message delivery. For those who also want to see what is happening on the other side of the antenna, InSight Desktop's industry-leading ZigBee protocol analysis and visualization engine makes a perfect companion to the Training application. What packets are sent over the air after a call to `emberSendDatagram`? What does 802.15.4 association look like? What route is this message taking? How and when are routes discovered? With the Training application and InSight Desktop, the answers are literally at your fingertips; just type in the command and see exactly how it all works.

This chapter uses the Training application as a way to learn about ZigBee networking concepts and the EmberZNet development environment. It takes a topic-based approach, and introduces commands to illustrate concepts.

Getting Started

In this section you will use the Training application to create a two-node ZigBee network and send data over the air.

Setup

The Training application software is provided for the host as a pre-built binary image. For the EM260, the Training application runs on a PC host running cygwin (www.cygwin.com) and communicates with the EM260 over the serial UART. In this case, the EM260 must be running the image that supports EZSP over UART. The EM260 Training application also runs on the Atmel AVR ATmega128 host processor and

communicates with the EM260 network coprocessor via the high-speed serial interface (SPI). When using the Training application version for the Atmel AVR ATmega128, the EM260 must be running the image that supports EZSP over SPI. The command line interface for the two versions is identical.

Using InSight Desktop (ISD), upload the appropriate EZSP firmware to three nodes. To do this, first load the correct image on the EM260 network coprocessor (NCP): right-click on the adapter icon in the Adapters view, select **Connect**, then select **Upload network coprocessor firmware**, and then select the correct image. To load the Training application on an Atmel AVR ATmega128, select **Connect**, then select **Upload host application...**, and then select the Training application built for the Atmel AVR ATmega128 (build/training-host-avr128.bin).

Note: If you select all three adapters at once before right-clicking, you can launch the uploads simultaneously.

The Training application listens for commands and prints responses on the command line interface. (If using the ATmega128 host with the EM260, the interface is provided on serial port 1, and the UART settings are 19200 8-N-1, with no flow control. EZSP UART hosts must specify serial parameters as arguments to the training host executable. Run the executable with the `-?` option for a list of available arguments.

For the ATmega128 host, there is no need to start pulling out serial cables, however, because the InSight Adapter passes serial port 1 activity through to TCP port 4901. An ISD plugin provides the ability to open a terminal session directly within ISD: right-click on the adapter (in the Adapters view) connected to the target node and select **Launch Console**. Alternatively, you can use any telnet program.

Note: Telnet programs that buffer input a line at a time work best because they allow backspacing to fix typos. PuTTY is a free telnet/ssh client that does this by default.

You can use ISD to determine the IP address of each adapter by clicking on the **+** symbol next to the adapter icon to expand its properties. Then use your telnet program to open a connection to port 4901 (for example, type `telnet` *`[ip address]`* `4901`) for each node.

For the EZSP UART setup, a Cygwin or Linux host computer must be connected to the EM260 board through the physical serial interface (DB-9 or USB, depending on what your breakout board supports). Refer to your breakout board's technical specifications (120-2006-000 and 120-2007-000) for information about how to configure the serial interface appropriately for a physical connection (rather than through the DEI cable).

To verify that you are successfully connected, type **help** or **h** and press Enter. If all is well, you will see the Help menu, organized by sections. If not, the serial port settings may be wrong. The InSight Adapter comes factory-configured with the correct serial port settings for connection via the ATmega128 host. If they have been changed, telnet to port 4902 (the admin port) and type `port 1 19200 8-N-1`. If using the AVR ATmega128 for your EZSP host, be sure that the DEI cable is connected and that the breakout board is configured appropriately to use DEI as its serial interface. If using EZSP-UART, the default settings to connect to the NCP should be 115200 bps, no flow control.

Built-in help

Type `help` or `h` to see the Help menu. It looks like this:

```
help or h
    int8u section
        0 CONFIGURATION
        1 UTILITIES
        2 NETWORKING
        3 BINDING
        4 MESSAGING
```

```
        5 QUICK
```

The line `int8u section` means that the help command takes an integer argument for the section. Below that, the section names are listed next to their number. Each section contains a group of commands. For example, to see the binding commands type `help 3`:

```
BINDING COMMANDS

clearBindingTable
setBinding
getBinding
deleteBinding
bindingIsActive
getBindingDestinationNodeId
setBindingDestinationNodeId
```

Typing the name of any command prints the usage for that command if it has parameters, or executes the command otherwise. For example, type `setBinding` to see:

```
setBinding
    int8u index
    EmberBindingType type
        0 UNUSED_BINDING
        1 UNICAST_BINDING
        2 AGGREGATION_BINDING
        3 MULTICAST_BINDING
    int8u local
    int8u remote
    int8u clusterId
    EmberEUI64 identifier
```

Each parameter's type and name is listed on a line by itself. If the parameter uses named constant values, the names and their values are printed out for easy reference. Use the numerical values when typing the command.

Command syntax

To execute a command, type its name followed by its parameters, separated by spaces. If you make a mistake in the type or number of parameters, the Training application will politely inform you by printing out the command usage. Case is ignored (so you can type in all lowercase or all uppercase if you feel like it).

Integer parameters may be specified in decimal or hexadecimal format (using the "0x" prefix), whichever is more convenient. A byte array such as an EUI64 is written as a string of two-character hex bytes with no "0x" prefix and optional whitespace, enclosed in curly braces. For example, to set a unicast binding at index 0, local and remote endpoints 5, cluster id 0x7E, and EUI64 as follows, type:

```
setBinding 0 1 5 5 0x7E {00 0D 6F 00 00 06 6C 98}
```

You can also specify a byte array in ASCII by enclosing it in double quotes, as shown in the section Sending a message. Finally, for technical reasons, a parameter of type int32u must be entered as a 4-byte hex array (high byte first) rather than as an integer. (This only affects the `channelMask` parameter of the scan commands.)

Forming a network

Now for some fun! Let's create a ZigBee network. A ZigBee network consists of a single coordinator, any number of routers, and any number of end devices.

As its name suggests, a *router* is a node that participates in relaying packets through the network. It is always powered and cannot turn off its radio to conserve power.

An end device does not route traffic. Instead, it chooses a router as a parent and can communicate with other nodes only through its parent. An end device typically turns off its radio whenever possible

to conserve power, in which case we call it "sleepy." (The ZigBee specification uses the more laborious phrase "rxOffWhenIdle is true".)

The coordinator is just a router. There is nothing special about a coordinator except that according to the ZigBee specification it must have node id 0x0000 and it must be the first node to start a network. To avoid writing "router or coordinator" constantly, in this chapter the term "routers" will include the coordinator unless specifically indicated to the contrary.

To start a network, choose one of your nodes to be the coordinator. We will tell it to become the coordinator (known as "forming a network" in ZigBee parlance) with the `formNetwork` command. First, type `formNetwork` to see the list of parameters:

```
formNetwork
    int16u panId
    int8s radioTxPower
    int8u radioChannel
```

The PAN id is a two-byte value used to distinguish multiple ZigBee networks on the same channel from each other. The radio power is in units of dBm and has a valid range of -43 to 3 for the EM250 and EM260 radios. The channel is one of the 802.15.4 channels in the 2.4 GHz spectrum, which are numbered from 11 to 26.

For example, type `formNetwork 0x1ABC -1 11`. This tells the node to become a ZigBee coordinator using PAN id 0x1ABC, radio power -1dBm, and 802.15.4 channel 11. In EmberZNet API terms, the command simply calls `emberFormNetwork()` with the corresponding arguments. Barring any typos, you'll see this response:

```
formNetwork status:SUCCESS
stackStatus status:NETWORK_UP
```

The first response line indicates that the `formNetwork` command executed successfully. The second line is a callback indicating that the formation process is complete and the stack is up and ready. It corresponds to the `emberStackStatusHandler()` callback.

Watching the action

Now is a good time to start watching what packets are flying around using InSight Desktop. Right-click on the coordinator's adapter icon in the Adapters view and select **Start capture...**. This causes ISD to collect, analyze, and display all of the coordinator's radio traffic in the editor window labeled **Live**.

In the Event log you'll see that the coordinator is periodically transmitting a "Neighbor Exchange" packet. You may wish to hide these packets to make it easier to see other packets. Select **Filters | Hide Neighbor Exchange** from the menu bar or by right-clicking in the editor pane.

Joining a network

In ZigBee and 802.15.4, nodes other than the coordinator must ask to join an existing network for the first time. The joining node performs an "active scan" to find a router, then asks it to join using the 802.15.4 association procedure. If successful, the router assigns the joining node a two-byte network id (also known as a *short id*).

Don't worry if this sounds complicated. The EmberZNet stack performs the whole procedure automatically via the `emberJoinNetwork()` API call, which is exposed in the Training application as the `joinNetwork` command:

```
joinNetwork
    EmberNodeType nodeType
        1 COORDINATOR
        2 ROUTER
        3 END_DEVICE
        4 SLEEPY_END_DEVICE
```

```
        5 MOBILE_END_DEVICE
    int16u panId
    int8s radioTxPower
    int8u radioChannel
    boolean joinSecurely
        0 FALSE
        1 TRUE
```

In addition to the parameters used to form the network, you must also specify the node type and whether to join securely or not. These two parameters both use named constant values. To join as a router, type `joinnetwork 2 0x1ABC -1 11 0`:

```
joinNetwork status:SUCCESS
stackStatus status:NETWORK_UP
```

The `joinNetwork` response is returned immediately. The `stackStatus` callback follows the completion of the over-the-air joining process (in the EmberZNet API it is called the `emberStackStatusHandler()`).

The Live capture window of ISD shows what happened over the air. A beacon request (sent by the as-yet-anonymous joining node) elicited a beacon from the coordinator; that was the active scan. It was followed by an 802.15.4 association, which consisted of six packets.

You can examine the frame format of each packet by clicking on its row in the ISD event table and looking at the event detail pane. The whole association transaction is also summarized in a single row of the transaction log.

Sending a message

Now you will actually send some data through the network. Of the many possible options (e.g., `sendBroadcast`, `sendUnicast`, `sendDatagram`, etc.), you will start with the `unicast` command. This is a simplified version of the `sendUnicast` command, in which most of the arguments are filled with default values. It calls the `emberSendUnicast()` API.

```
unicast or u
    EmberNodeId destination
    bytes messageContents
```

Use this command to send a message to the coordinator from the other router joined to the network. In ZigBee networks, the coordinator always has id 0. So type:

```
unicast 0 "EmberZNet is easy!"
```

The sending node prints the following two responses:

```
sendUnicast status:SUCCESS
unicastSent destination:0x0000 profileId:0xC00F clusterId:1
  sourceEndpoint:1 destinationEndpoint:1 options:APS_RETRY |
  ENABLE_ROUTE_DISCOVERY messageTag:1 status:SUCCESS
```

The first response indicates that the call to `emberSendUnicast()` returned successfully. The second response reports the results of the `emberUnicastSent()` callback, which is invoked upon receipt of an end-to-end acknowledgment from the coordinator.

Meanwhile, the coordinator reports the arrival of the incoming message:

```
incomingMessage type:UNICAST profileId:0xC00F clusterId:1
  sourceEndpoint:1 destinationEndpoint:1 options:APS_RETRY |
  ENABLE_ROUTE_DISCOVERY lastHopLqi:240 lastHopRssi:-72 sender:0x48A0
  bindingIndex:255 datagramReplyTag:0 messageLength:12 messageContents:
  20 45 6D 62 65 72 5A 4E 65 74 20 69 73 20 65 61 73 79 21
  " EmberZNet is easy!"
```

The contents of the message are printed in hex and ASCII.

In this section you formed a network and sent a message, so you have officially gotten started.

Anatomy of the Training Application

This section discusses the relationship between the Training application and the EmberZNet API, and gives a high-level overview of the command set.

Relationship to EZSP

The EmberZNet Serial Protocol (EZSP) is a binary protocol used to control the EM250 SoC platform or EM260 network co-processor from a host microprocessor (over a SPI or UART). Its command set was carefully designed to provide complete access to EmberZNet networking functionality.

The Training application is conceptually similar to EZSP in that it provides control of an EmberZNet node via a serialized command interface, so it was natural to leverage EZSP in the design and implementation of the Training application.

The Training application command set is really just the EZSP command set—the Training application simply translates the commands from human-readable ASCII text into binary EZSP frames and then delivers them to the EZSP processing engine. It then translates the binary EZSP response frames back into human-readable ASCII text and prints them out. The Training application is, in essence, just a glorified EZSP translator.

As a result, the Training application is the ideal tool for users of the EM250 or EM260 development kit to learn EZSP through hands-on interaction. And since EZSP is largely a direct serialization of EmberZNet API calls, it is also an ideal tool for users of the EM250 dev kit to learn the EmberZNet API and networking concepts without the overhead of writing and compiling code.

Command groups

As mentioned earlier in this chapter, the command set is divided into six groups: configuration, utilities, networking, binding, messaging, and quick. The first five groups contain all the EZSP commands, organized and named exactly as they are in the *EM260 Datasheet*. The "quick" group contains convenience commands specific to the Training application.

With a few exceptions, the networking, binding, and messaging commands are direct serializations of EmberZNet API calls. The EmberZNet function name is obtained by prepending "ember" to the command name, as you saw earlier with `formNetwork` and `emberFormNetwork()`s.

The configuration commands are EZSP-specific and do not have direct counterparts as EmberZNet API functions, because configurations are typically performed statically in an EM250 application, but must be performed at runtime when using a self-contained network coprocessor. The utilities commands are also EZSP-specific; they make some of the hardware resources of the network processor, such as RAM, timers, and EEPROM, available to the host processor. The EZSP host API function name corresponding to these commands is obtained by prepending "ezsp" to the command name.

The quick commands are specific to the training application and are not part of the EZSP command set or the EmberZNet API. The `unicast` command is a simplification of `sendUnicast`, which uses default values for many parameters to save you some tedious typing. `unicastOptions` is used for setting one of those defaults. The `stream` command is a simple utility that sends a specified number of APS unicasts as fast as possible.

Differences from EZSP

For the assiduous student of EZSP, there are a number of minor technical differences between the Training application and EM260 interfaces (apart from the obvious ones).

The Training application inherits the EM260's default configurations. To minimize the amount of initial typing needed to get started, it performs the following configurations after a reset:

- Adds a default endpoint 1 with profile id 0xC00F, device id 0x1234, and input and output cluster id 0x0001. It is used as the default endpoint for the quick commands `unicast` and `broadcast`.

- Sets the security level to 0 (no security).
- Sets the encryption key to `ember EM250 chip` (as a 16-byte ASCII array) for convenience when security is enabled.
- Turns the 802.15.4 permit joining flag on.

There is no mechanism in the Training application for sleeping and waking the microprocessor as there is for the EM260. (The radio does sleep and wake for sleepy end devices, but not for the processor.)

The only EZSP policy setting available in the Training application is the trust center policy. The other policies are not applicable for a user-driven application.

8 ZDO Sample Application

The ZDO Sample application is a ZDO (ZigBee Device Object) application using the EmberZNet stack. This application builds a two-node network and lets you send ZDO messages using a command line interface.

Expected Operation and Data Flow

This application does not create a network larger than two devices. The data flow is between these devices using standard ZigBee ZDO commands.

A ZDO is specified within the application layer of the ZigBee specification. The ZDO represents a base set of functionality to provide an interface between ZigBee devices to allow device discovery, security management, network management, and binding management. The applications use these underlying ZDO commands during the startup and operation of ZigBee networks. Because these functions are required in the underlying ZigBee stack, applications can use these functions in mixed networks to ensure interoperability.

Notes and limitations

This application is supported on the EM250, EM260, and EM2420.

This sample application demonstrates ZDO command functionality only and is not intended as a scalable network sample application.

This application only uses one endpoint for simplicity. You can modify it to use other endpoints or multiple endpoints.

For ease of use, the devices in this sample application are routers.

Application Interface and User Operation

All user interface work is done using a command line interface on the serial port. The serial port configuration for this application is 115200 baud for the EM250. For other devices supported the serial port operates at 38400 baud.

This application provides some simple management commands that can be sent to the node via the serial port. Table 6 lists these commands.

Table 6. Management Commands

Command	Description
help	Prints the Help menu
version	Prints the version of the application
info	Prints information about this node including channel, power, and application
network	Network commands: `form`, `join`, `leave`, `permit join`
zdo	Sends ZDO commands (see the “ZDO Commands” section)
print	Prints the binding table
reset	Resets the node

To start a network, use the `network form` command. This command uses channel, power, and the PANID as arguments. The second device can then join using the `network join` command. To allow the second device on the network, `permit join` must be on. Once a network is formed, you can use ZDO commands between the devices.

ZDO Commands

The following abbreviations are used in the ZDO commands:

- src = source
- dst = destination
- sep = source end point
- dep = destination end point
- tep = target end point
- rep = requested end point

The following sections categorize the ZDO commands. Table 7 clarifies some of the command fields.

ZDO device and discovery attributes commands

```
zdo netAddrReq        <sep: 1 int> <dstEUI:8 Hex>
                       <OPT kids: 1 str, def=T> <OPT idx: 1 int, def=0>
zdo ieeeAddrReq       <sep: 1 int> <dstId:2 Hex>
                       <OPT kids: 1 str, def=T> <OPT idx: 1 int, def=0>
zdo nodeDescReq       <sep: 1 int> <dstId:2 Hex>
zdo nodePwrDescReq    <sep: 1 int> <dstId:2 Hex>
zdo nodeSmplDescReq   <sep: 1 int> <dstId:2 Hex> <tep: 1 int>
zdo nodeActvEndPntReq <sep: 1 int> <dstId:2 Hex>
zdo setInClusters     <num> <clusters>
zdo setOutClusters    <num> <clusters>
zdo nodeMtchDescReq   <sep: 1 int> <dstId:2 Hex> <profile int>
```

ZDO bind manager attributes commands

Note: The source is the device you are adding the remote binding to, and the destination is where that device is meant to communicate to.

```
zdo endBindReq  <tep: 1 int>
zdo setEndPnts  <sep: 1 int> <dep: 1 int> <rep: 1 int>
zdo setBindInfo <clusterId 2 Hex> <type int> <grpAddr: 2 Hex>
zdo bindReq     <dstId:2 Hex> <srcEUI:8 Hex> <dstEUI:8 Hex>
zdo unBindReq   <dstId:2 Hex> <srcEUI:8 Hex> <dstEUI:8 Hex>
```

ZDO network manager attributes commands

```
zdo bindTblReq    <sep: 1 int> <dstId:2 Hex> <idx: 1 int>
zdo leaveReq      <sep: 1 int> <dstId:2 Hex> <dstEUI:8 Hex>
                  <leaveReqFlgs int>
zdo nwkUpdateReq chan <chan:1 int>
zdo nwkUpdateReq scan <dstId:2 Hex> <dur:1 int> <cnt:1 int>
zdo nwkUpdateReq set  <mgrId:2 Hex> <mask:4 Hex>
```

Sample interactions

Table 7 lists some examples of typical interactions to set up a network and execute ZDO commands.

Table 7. Node Interactions for Setting Up a Network

Node 1 Command **NodeId: 0000** **Eui64: 000D6F00000E5403**	**Node 2 Command** **NodeId: 0EDC** **Ei64: 000D6F00000B4562**	**Description**
`network form 15 -6 1025`		Node 1 forming the network
`network pjoin 255`		Node 1 turning on permit joining
	`network join router 15 -6 1025`	Node 2 joining the network
`zdo netAddrReq 1 000D6F00000B4562`		Node 1 requesting the short address of Node 2
`zdo ieeeAddrReq 1 0EDC t 0`		Node 1 requesting the long address of node 2
`zdo nodeDescReq 1 0EDC`		Node 1 sending node descriptor request
`zdo bindReq 0EDC 000D6F00000B4562 000D6F00000E5403`		Node 1 sending a binding request to Node 2
`zdo leaveReq 1 0EDC 0000000000000000 10`		Node 1 sending a leave request to node 2