

Introducing AppBuilder

For the EM250 SoC Platform and EM260 Co-Processor

Contents

Introduction	2
What Is AppBuilder?	2
What Are Clusters?	2
What Is the ZCL?	6
ZigBee Application Profiles	6
Application Profiles Enable AppBuilder	7
The AppBuilder GUI	10
Creating a Network	11
Sending a Message	12



Introduction

This document presents the following topics:

- What is the AppBuilder?
- What are the goals of AppBuilder?
- How does AppBuilder work?

In order to understand how AppBuilder works, it is important to understand ZigBee clusters and ZigBee application profiles, which are described in this document.

What Is AppBuilder?

AppBuilder is a tool for generating ZigBee-compliant applications. AppBuilder is made up of two parts: a super set of all source code needed to develop any Smart Energy or Home Automation compliant device, and a graphical tool for configuring the included source code. The source code provided with AppBuilder includes:

- Implementations of ZigBee clusters
- Parsing code for incoming ZCL messages
- Serial command line interface (CLI)
- Example main.c

The AppBuilder graphical tool is both a standalone application and an InSight Desktop plugin. The AppBuilder provides an interface to the user for turning on or off embedded clusters and features in the code compiled into a finished application.

The Goals of AppBuilder

AppBuilder is intended to meet the following goals:

- Ship ZigBee-compliant sample applications based on the ZigBee Home Automation (HA) application profile and the ZigBee Smart Energy (SE) application profile with EmberZNet for the EM250 and EM260 platforms.
- Enable rapid development and decrease customer time-to-market by providing standard SE and HA applications.
- Assist customers in configuring stack and HAL settings in custom or compliant applications.

What Are Clusters?

Before using AppBuilder, it is important to understand how Zigbee Clusters work. Each Zigbee cluster defines an application level protocol. A set of these protocols (or clusters) defines the functionality of a particular ZigBee device. Anyone with a networking background can think of a cluster as an application protocol that has been encapsulated within the Zigbee specification.

Clusters specify two things: attributes and commands. Attributes are well-defined pieces of data that are stored on a device and can be read (and sometimes written) by external devices. Commands specify over-the-air messages that are exchanged. Each command defined by the Zigbee Cluster Library is unidirectional in the sense that it is sent by one side (either the client or server) and received by the other. A device can implement only one side of a cluster, or it can implement both sides of a cluster. For instance, an "HA on/off Light" implements the server side of the "on/off" cluster, while the "HA on/off Light Switch"

implements the client side of the “on/off” cluster. This defines that the Switch sends “on”, “off”, and “toggle” commands that the Light is capable of receiving (and understanding). It also defines that the Light stores a Boolean attribute called “on/off” representing the current state of the device.

Example: The Identify Cluster

The Client/Server interaction defined by the ZCL is illustrated in Figure 1.

Figure 1. Cluster example: Identify

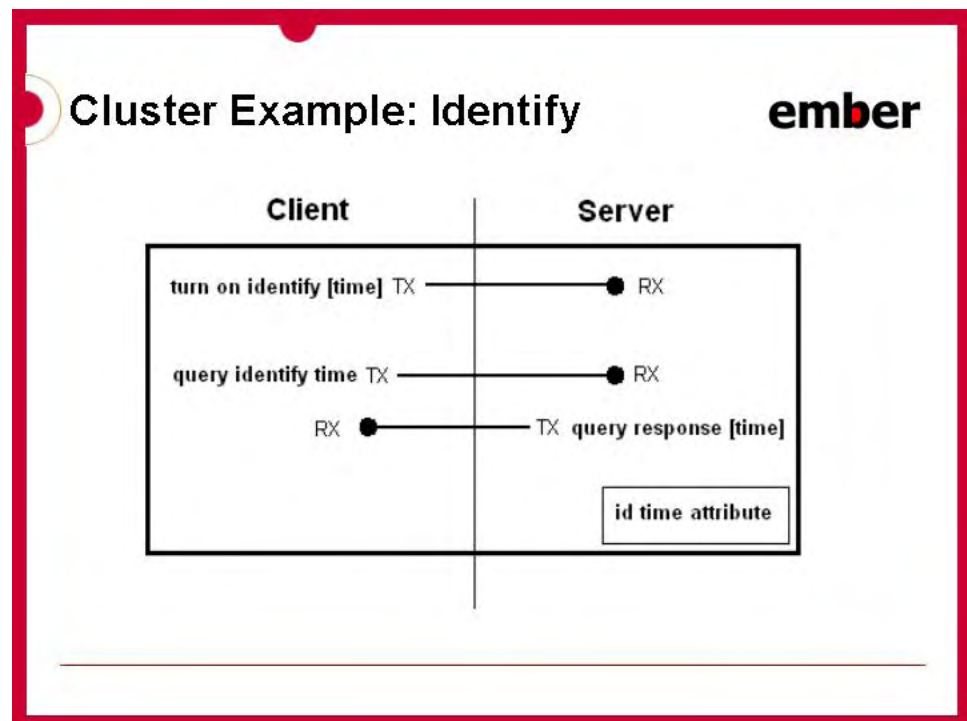


Figure 1 shows the Identify cluster. Like many clusters, the Identify cluster is a fairly simple cluster. The lower right corner shows the single attribute: identify time.

Identify cluster use case: A user is provisioning a network of 12 lights in one room and must connect 6 of those lights to a single switch. The MAC address of each light will be used to associate it with the switch. The MAC addresses for all 12 lights can be discovered by using a provisioning tool and a low power broadcast or by using a token (set when they were installed) on each light indicating room or location. The “Identify” functionality can be used to figure out which six MAC addresses correspond to the six physical lights that the user wants bound to a switch. Using the Identify cluster, the user can tell each light individually to “identify” itself (for example, blink so that it can be seen).

The Identify cluster defines the protocol for how devices are put into and taken out of identify mode. In the above example, the provisioning tool implements the client side of the identify cluster, and the light or the device that needs to be identified implements the server side.

When the client wants to tell a device to “start identifying,” it sends the “Identify” command and specifies a period of time in seconds for which to continue identifying. The device stops identifying when the identify time attributes (decremented each second) reaches 0, or if the device receives an “Identify” command with identify time value of 0.

The first message in Figure 1 turns on “identify.” When identify is turned on, a time period is also included in the message. For example, suppose identify is turned on for 30 seconds.

The second message in Figure 1 shows the client (provisioning device) querying the server (light) to find out how much time is left in the identify process.

Because a query message can be sent to a group, it is possible to put a device into a mode where it is identifying, and then use a PC or provisioning tool and figure out which device in the group is identifying. This is useful if a device supports a physical cue to start identifying. Then a device can be poked (button press, magnet wand, and so on) to start identifying, and a group message can be sent to map the MAC address to the physical device.

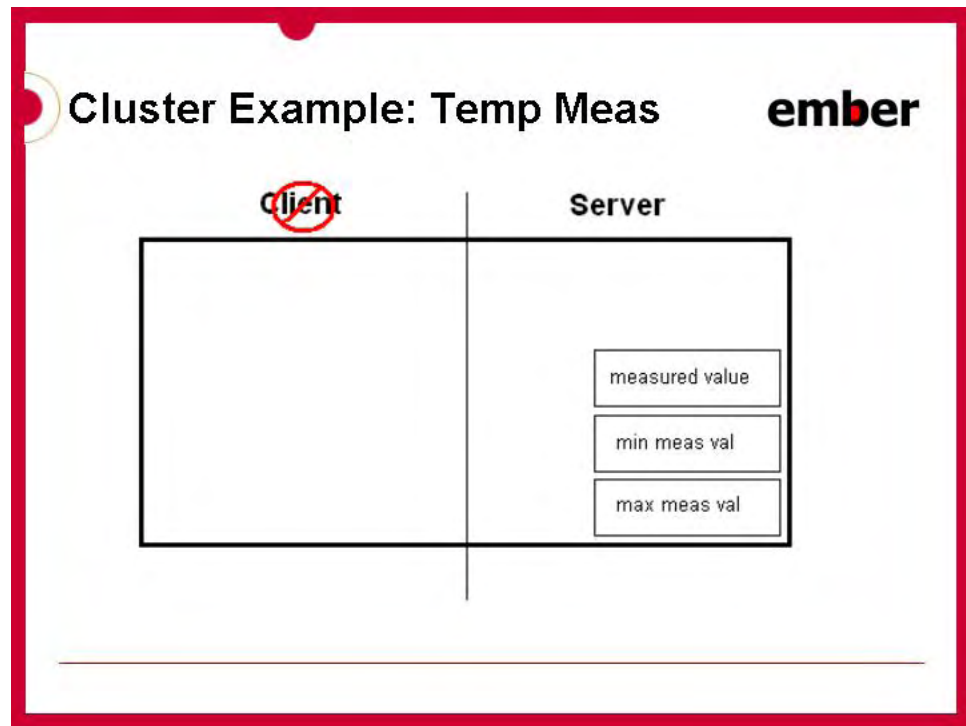
Note: ZigBee often uses the terms “in-cluster-list” and “out-cluster-list” instead of server and client. An “in-cluster-list” is the list of supported server clusters, and the “out-cluster-list” is the list of supported client clusters.

In most cases, the server side of a cluster contains all the attributes, and the client side is the side that initiates an over-the-air exchange. For the most part, the client sends a message, and the server answers that message.

Example: The Temperature Measurement Cluster

Figure 2 illustrates another example of a cluster. This example shows temperature measurement.

Figure 2. Cluster example: Temperature measurement



Notice that this cluster has no commands—it only has attributes. In this case, the device implements measurements of temperature, such as a thermostat. This example includes a measured value, a minimum measured value, and a maximum measured value. With no commands, this cluster relies on the global commands defined in the ZCL. The global commands define messages for reading, writing, discovering, and reporting attributes.

Global Commands: There are 14 global commands that read attributes, write attributes, configure attribute reporting, discover attributes, and report attribute values. Clusters that only include attributes are simple to understand and simple to implement, because the global commands are already implemented.

In order to read the value of an attribute of this cluster, a global read attributes command is used. This message contains the attribute ID of the attribute to read. In combination, the cluster and the attribute ID provide unique identification. On the embedded side, this makes it possible to centralize all the attributes in a single table. All of the code for those attributes is generic, shared code.

As a result, for example, when adding four of the temperature-measuring-sensing clusters, the impact on flash is minimal, because there are no additional commands. The impact on RAM depends on the number of attributes added per cluster.

The application level protocol provided by the Zigbee Cluster Library makes it possible for two companies to develop products separately and have them work together without having to test them together.

What Is the ZCL?

ZCL stands for the ZigBee Cluster Library. It is a document that specifies the mini-protocols (clusters) used by ZigBee devices.

The original ZCL document had 30 clusters, most of which were specified as required or optional by at least one device in the ZigBee HA application profile. The SE application profile uses some of the clusters specified in the ZCL but also specifies new clusters that are unique to SE.

ZigBee Application Profiles

ZigBee application profiles specify generic settings (such as security, join parameters, and poll rate) for all devices within an application group. Application profiles also specify exactly what clusters (protocols) must be supported for each device in the application group. A ZigBee device can be thought of as a collection of clusters.

There are currently two ZigBee application profiles supported by AppBuilder:

- Home Automation (HA)
- Smart Energy (SE, formerly AMI or Automated Meter-reading Infrastructure)

For example, an on/off switch and an on/off light are 2 of the 31 devices in the Home Automation profile. All of the devices within a profile must use the same sort of security. There are recommendations on polling rates, start-up parameters, what kind of ZDO messages should be implemented, and so on, with the idea being that these devices must interoperate together on the same network. If devices have different security settings, they cannot join together. Therefore, if a user buys an HA device from company A and buys an HA device from company B, one of the devices should be able to join the other device.

If two ZigBee devices are on a certified ZigBee stack, they can route for each other. In other words, they can exchange messages at the application level. There is no guarantee of interoperability at the application level until they use an application profile. It is these standard application profiles that enable AppBuilder to generate compliant ZigBee applications.

Examples of Devices

One example of a device is an HA on/off light with the following implementations:

- Identify server (required by all)
- Groups server
- Scenes server
- On/Off server

Another example of a device is an HA on/off light switch with the following implementations:

- Identify client
- Groups client
- Scenes client
- On/Off client

The on/off light switch can send an on/off or toggle message that the on/off light is required to understand and abide.

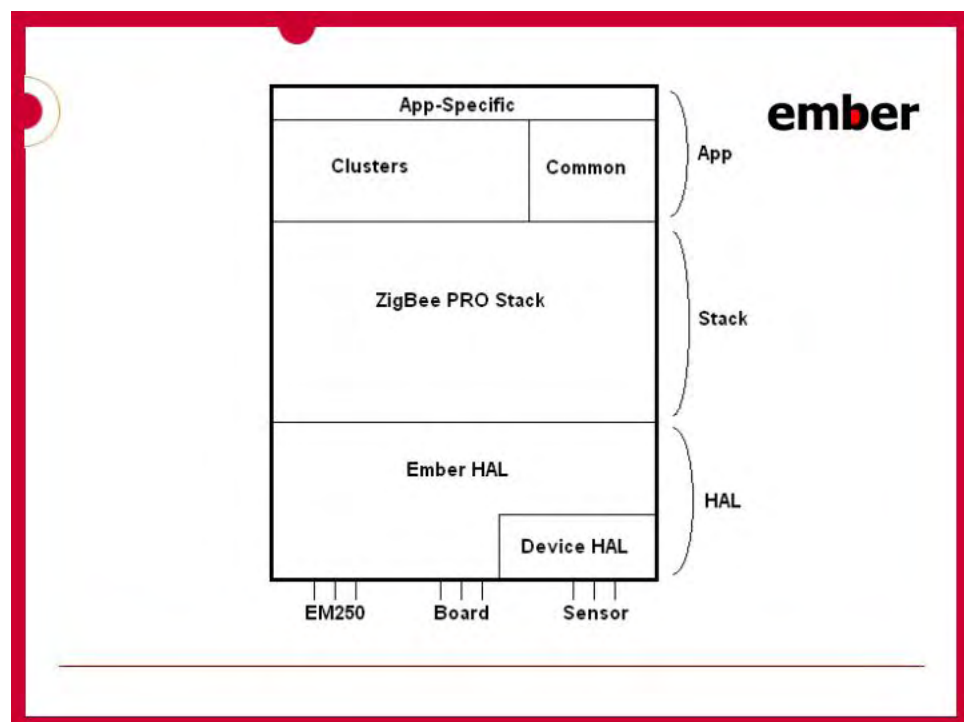
Application Profiles Enable AppBuilder

The ZigBee application profiles have enabled Ember to create AppBuilder. Ember's customers appreciate receiving as much code as possible to help them customize their products, because they know exactly what their device needs to do. For example, when a company develops an in-home display, that company should be very good at developing a nice display, which might include the best touch-screen display, the best graphics, the lowest cost, or the best appearance. Of course, none of these issues have anything to do with the data transmission software.

As a result, the more embedded software that Ember can provide to its customers, the better chance the customer has of being successful because they can combine their expertise with Ember's expertise. With the GUI AppBuilder and the embedded code, customers can now create their own embedded code.

Some important questions include: How much of this code can AppBuilder create? Where does it fit in? What do customers have to do? To answer these questions, consider Figure 3. The right side of the figure shows the application, the stack, and the HAL.

Figure 3. The application, the stack, and the HAL



The stack is a compiled piece of code that Ember gives to customers. It is not provided as source, and as a result, customers cannot change that code. If there is a problem in the stack, customers ask Ember to fix it.

Ember provides a reference HAL that works on the EM250 on Ember's development kit. The reference HAL is provided to customers as source code, because they need to modify it and make it work with their sensors with the way their board is set up and with the way they use the GPIOs and the EM250. In other words, customers will need to tweak the Ember HAL.

There may be another piece of the HAL that they have to make device-specific in the case of a special sensor: They have to use an example provided by Ember.

The Cluster, Common Code, and Application Specific Pieces

Under the application in Figure 3, there are three segmented pieces: cluster, common code, and application specific. The cluster code is all the code needed to implement the ZigBee clusters, which are the clusters that make a specific device and application profile. This is provided by Ember as part of the HA- or the AMI-embedded application.

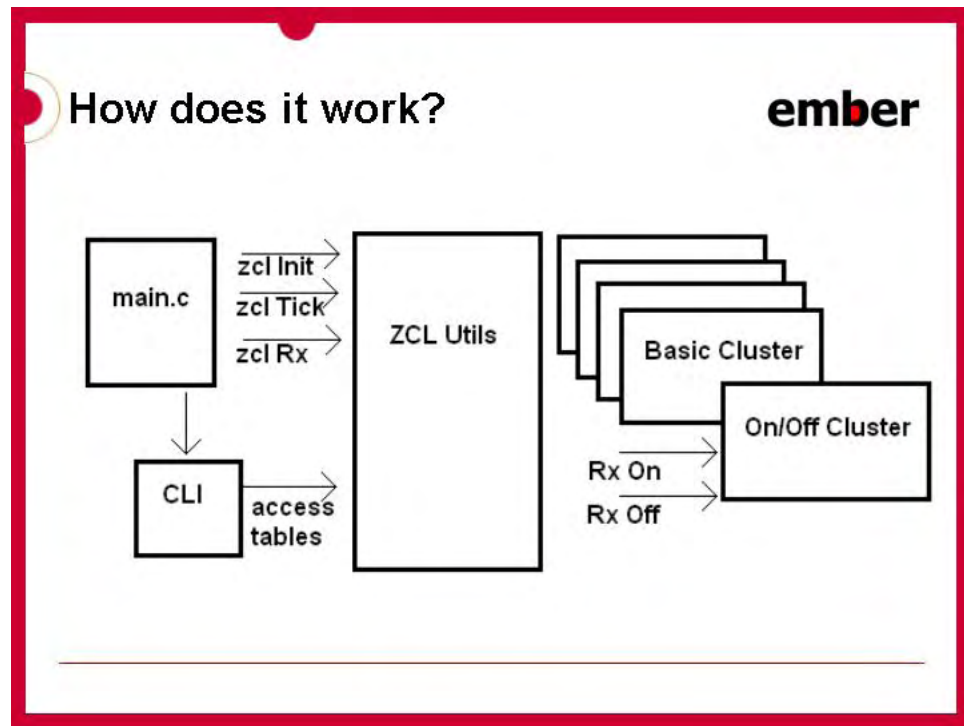
The common code is also provided for the application. In Figure 3, the common code is separated from the cluster, because the cluster must be available as a library for users who already have existing applications. In this case, it is possible to add the ZigBee cluster library to existing applications. Users can add clusters of functionality with three or four API calls. The "common" section in Figure 3 the above figure can be replaced with a customer's application.

At the top of Figure 3 is an application-specific piece that can shrink or grow. For example, suppose a control for the application-specific piece takes up most of the application section in the figure. As a result, the other pieces will shrink. On the other hand, if a customer does not want to perform software development, the application-specific piece will shrink to nothing.

Typically, users will have some kind of application-specific detail in terms of tying cluster functionality into their hardware. While there is likely to be some work involved, with the device HAL, it might be an issue of connecting to one of the clusters. For example, pushing a button may change the temperature, which is an attribute in the cluster. If the device is on the client side where there are messages being sent, then user interface work will need to be done with regard to determining how to get that information into the device in order to send the message.

As already described, there is a cluster piece and a common piece in Figure 3. ZCL Utilities is a large part of the cluster piece. As shown in Figure 4, the ZCL Utilities and the right side of the diagram represent the cluster.

Figure 4. How does it work? (The cluster)



Notice the main.c and a command line interface (CLI)—that is the common piece. There are three entry points to the ZCL Utilities:

- Init
- Tick
- Receive

Users can substitute their own application. A user either calls `init` one time or calls `tick` as often as possible with the `emberTick()` function. It is necessary to call `receive` when receiving an incoming message. The tick call allows clusters to perform tasks periodically. Therefore, it is possible to identify a cluster as every second ticks down. Receive allows the utility to have a chance of processing the message. The utility returns a Boolean value and notes whether the message has been processed so users know whether to take it as part of their application.

The clusters are broken up into the separate files. The ZCL Utilities perform all of the incoming parsing of messages. This is centralized, because if this task were broken up by cluster, it would result in code redundancies.

Once a message is parsed, a function is called. All of the function calls that receive messages are in the clusters. The “on/off” cluster will have an on/off init, an on/off tick, an on/off receive on, an on/off receive off, and an on/off receive toggle. The reason for this is to provide customers with the ability to customize their application. If they want to replace the on/off cluster, they have to replace five functions that are in that file. They can remove that file as their own file, which allows them to quickly and easily add their own code. This approach also prevents customers from touching the cluster decoding code.

Command Line Interface: The CLI allows the user to manually generate commands. It works on both the EM250 and the EM260. Most of the commands included in the CLI are used for crafting messages and sending them, which are tasks performed on the client side. The user has the ability to remove, replace or choose not to use the CLI.


Note: Figure 4 should include another block on the side for the application configuration. There is an application configuration file that controls the security settings and the clusters that are enabled. Obviously, it is not possible to have all of these clusters enabled because of the limitation on flash and RAM. The configuration makes it possible to choose the server clusters, the client clusters, the name of the device (which makes it possible for the prompt to be displayed as a name), the security level being run, the type of ZigBee device, the PAN ID, and the Extended PAN ID.

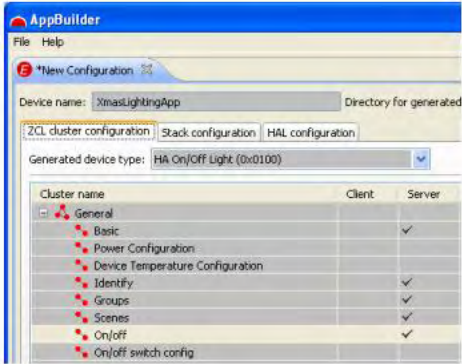
The AppBuilder GUI

The GUI does not generate any code—all the code already exists. The code is protected with `# define`, as shown in Figure 5.

Figure 5. How does it work? (GUI)

How does it work? (GUI)





```
#define BASIC_CLUSTER_SERVER
#define IDENTIFY_CLUSTER_SERVER
#define GROUPS_CLUSTER_SERVER
#define SCENES_CLUSTER_SERVER
#define ON_OFF_CLUSTER_SERVER

#define HA_PROMPT XmasLightingApp

#define DEVICE_ID 0x100
#define DEVICE_TYPE ROUTER
```

In Figure 5, the AppBuilder GUI is shown on the left. The configuration output from the GUI is shown on the right. The GUI generates a set of `# defines`, which is called a *generated configuration*. At build time, the compiled code includes the generated configuration file generated by AppBuilder. By using the `# define`, it is possible to turn on or off different features of the embedded code.

The user can choose a pre-defined device from the GUI that specifies an established set of clusters, or they can choose their own custom device. There are stack options for security, PAN ID, xPAN, and sleep time (that is, nap/hibernate). There are HAL options for the platform (EM250 or EM260), bootloader, debug level, serial port, and GPIOs.

Sleeping, for example, is one of the # defines. If a device is a sleepy device, the user gets all the code for a sleepy device. If not, all of the code for sleepy devices is excluded from the build. This approach is very modular. It is easy for users to determine when they have gone outside of their module if they try to build without a particular cluster, and another piece of the code is expecting that cluster to be there.

It is important for users to understand that they can choose a specific device or specific clusters. When a device is chosen, it specifies the exact clusters that should be used in order to conform to the Zigbee specifications. This is an easy way to ensure that devices conform to the ZCL specifications.

With regard to sleep time, there is a nap and a hibernate time. There are setups for napping time, which make it possible to send messages reliably to the device. Users can set up the PAN ID, security level, Extended PAN, and specify the preferred channels to use when performing join operations. A HAL section specifies the platform. For example, if users choose EM250, they can choose the bootloader, the debug level, and the serial ports they want to use, and specify how to configure the GPIOs.

Note: In order to have enough flash to support the device with the maximum number of clusters in ZigBee, it is necessary to use an old bootloader and no debug.

The maximum number of clusters per device tends to be three to seven clusters. Custom devices are allowed, but users should be aware that when custom applications are built, they could become too large. AppBuilder provides an estimate for how much flash will be used by the application as it is configured. This gives users the ability to make decisions based on what they need. For example, suppose a user chooses an old bootloader and learns that 10KB of flash is required. The user knows exactly how much space must be trimmed off their application in order for it to fit with all the functionality they require.

Creating a Network

Users can take two devices and start a network by using the Command Line Interface. The following commands form a network:

```
network form 11 2 00aa
```

Next, use the **pjoin** command to permit joining to make it possible for new devices to come into the network:

```
network pjoin 20
```

The second device would run the following **network join** command:

```
network join 11 2 00aa
```

Using these commands makes it possible for two devices to join with one network.

Sending a Message

This section presents two examples of sending a message. The first example is:

```
# read basic cluster (0) zcl version (0000)
zcl global read 0 0000
send 0000
```

The first command is a **zcl global read** command. This read command specifies cluster 0 and attributes 0. This is the basic cluster ZCL version. The second command is sending a read for the basic cluster ZCL version to the coordinator.

The second example is:

```
# identify for 30 seconds
zcl identify id 0030
send 0000
```

This ZCL command uses the Identify cluster. The identify command specifies identify time and abbreviated ID, and it sends a value of 30 seconds, which also goes to the coordinator.

When the user enters this command, it is loaded into a message buffer. When the command is built, the command line interface displays the contents of the message buffer for verification by the user. If the user makes a mistake in the command, that user has the opportunity to re-enter the command before sending it.

In order to send the **zcl** command over the air, the user must use a separate **send** command provided by the command line interface.

The **send** command has several additional options and endpoints that a user can specify. If it is broadcast, the user can send commands in groups.

The hash mark (#) in these examples indicates a comment. It is not typed into the parser.

Whenever a user sends a message, it tells the user which cluster is being transmitted. Likewise, whenever a user receives a message, it gives the user a printout of what it receives.

Debugging Levels

There are a couple of different levels of debug. All of the print messages within the library—the ZCL Utilities and all the clusters—and all the Ember serial print up calls are inside macros. There are two levels: an information level and a debugging level. Users can turn off all of their printouts by turning off these two macros. It is possible for users to turn off only the debugging messages and leave the information messages turned on.

For example, the information messaging mechanism can be used to print all of the tables in a specific cluster. Disabling all the print messages drastically reduces the size of the code, although the result is losing much of the ability to debug it.

Note: Messages are sent using the hard code node ID.

After Reading This Document

If you have questions or require assistance with the procedures described in this document, please contact an Ember support representative at support@ember.com.

Copyright © 2008 by Ember Corporation

All rights reserved.

The information in this document is subject to change without notice. The statements, configurations, technical data, and recommendations in this document are believed to be accurate and reliable but are presented without express or implied warranty. Users must take full responsibility for their applications of any products specified in this document. The information in this document is the property of Ember Corporation.

Title, ownership, and all rights in copyrights, patents, trademarks, trade secrets, and other intellectual property rights in the Ember Proprietary Products and any copy, portion, or modification thereof, shall not transfer to Purchaser or its customers and shall remain in Ember and its licensors.

No source code rights are granted to Purchaser or its customers with respect to all Ember Application Software. Purchaser agrees not to copy, modify, alter, translate, decompile, disassemble, or reverse engineer the Ember Hardware (including without limitation any embedded software) or attempt to disable any security devices or codes incorporated in the Ember Hardware. Purchaser shall not alter, remove, or obscure any printed or displayed legal notices contained on or in the Ember Hardware.

Ember, Ember Enabled, EmberZNet, EmberZNet PRO, InSight, and the Ember logo are trademarks of Ember Corporation.

All other trademarks are the property of their respective holders.

