UG102

EMBER® APPLICATION FRAMEWORK DEVELOPER GUIDE

The Ember application framework is a body of embedded C code that can be configured by Ember AppBuilder to implement any ZigBee Cluster Library (ZCL) application. This guide covers the structure and usage of the Ember application framework. Where appropriate, we have added information outlining differences between the latest release of the Ember application framework and its predecessors.

New in This Revision

Document renumbering.

Contents

1 Introduction

1.1 Purpose

The Ember application framework is a body of embedded C code that can be configured by Ember AppBuilder to implement any ZigBee Cluster Library (ZCL) application. The application framework is located in the app/framework directory.

This guide covers the structure and usage of the Ember application framework. Where appropriate, we have added information outlining differences between the latest release of the Ember application framework and its predecessors.

1.2 Building an Application

An application is created in several steps using the Ember application framework.

1. Create Ember application framework configuration files using Ember AppBuilder. The configuration files as well as the project files for your platform of choice are generated by Ember AppBuilder. An overview of using Ember AppBuilder and how it relates to the Ember application framework is provided in Chapter 17 of this manual. More detailed information on how to use Ember AppBuilder can be found in the Ember AppBuilder Help (Help | Help Contents for indexed help and Help | Dynamic Help for context-sensitive help).
2. Write the specifics of your application into the callback functions generated along with your configuration files. Use the Ember application framework API to do things like interact with attributes, and send, receive, and respond to commands on the ZigBee network. For more detailed information on the Ember application framework API, see Chapter 5, The Application Framework API.
3. Open the generated project file into the IDE of your chosen chip, compile your application, and load it onto your development kit hardware.
4. Run your application and interact with it using the Ember Desktop console window and the applications command line interface. More information on how to use Ember Desktop is available in the online help in Ember Desktop (Help | Help Contents).

1.3 Porting an Application

Information regarding the porting process is provided in the *Ember Application Framework Release Notes* (document number 120-8098-000A) included with your stack release.

2 Application Framework Architecture

The Ember application framework sits on top of the Ember stack, consumes the stack “handler” interfaces, and exposes its own more highly abstracted and application-specific interface to the developer.

One of the main features of the Ember application framework is the separation of user-created and Silicon Labs-created code. While Silicon Labs provides all of the source code for the Ember application framework, user-created code should live outside the framework and should interact with the framework through the Ember application framework API exposed by the framework utilities and callbacks. The block diagram in Figure 1 shows a high level overview of the Ember application framework architecture and how the two code bases are separated.

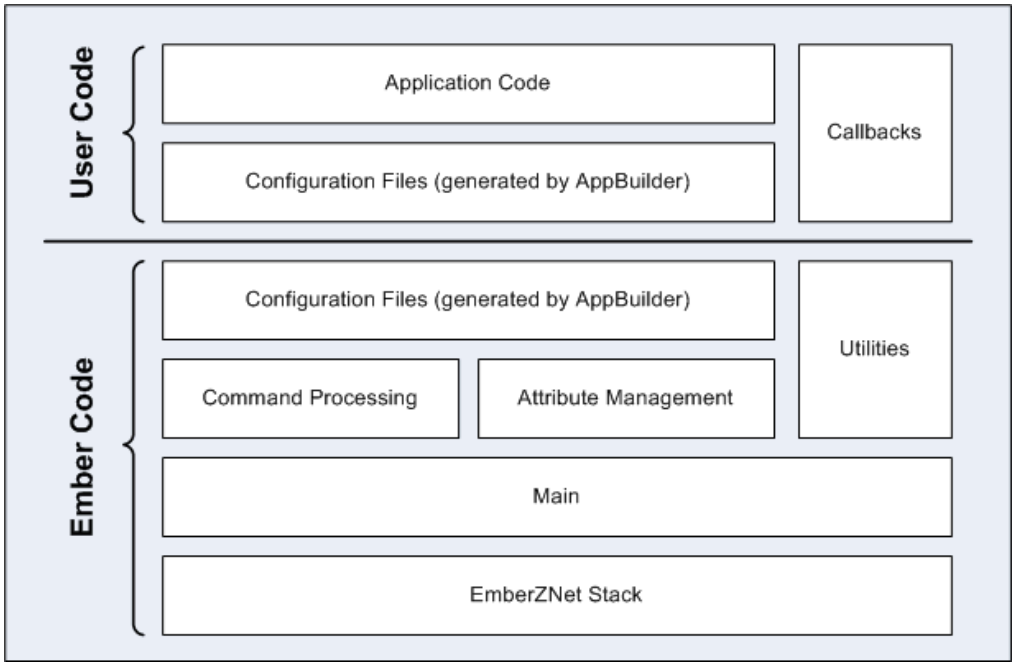

Figure 1. Application Framework Architecture

The main file included in app/framework/util consumes the Ember Stack handler interface and ties the Ember application framework into the EmberZNet PRO stack. Two main files are located in the app/framework/util directory, one (af-main-soc.c) for a System-on-Chip (SoC) like the EM250 or EM35x platforms and the other (af-main-host.c) for a host micro paired with a Network Co-Processor (NCP) like the EM260.

The main file implements the emberIncomingMessageHandler() and passes all incoming messages off to the Ember application framework for command processing. Once incoming messages are processed they are either passed off to the appropriate cluster for handling, or passed directly to cluster-specific callbacks generated by Ember AppBuilder. A significant portion of the command processing code is generated directly from the ZCL XML documents included in tool/appbuilder.

All of the code and header files generated from the ZCL XML documents are generated into app/builder/<application name>/ alongside the application header and callbacks file among others.

3 Application Framework Directory Structure

tool/appbuilder: Configuration and template files used by Ember AppBuilder

When you point Ember AppBuilder at a stack installation, it looks into this directory to load XML descriptions of the most current ZCL implementation as of the release of that stack.

Put your custom cluster .XML files in this location. For more information about creating custom clusters, see Ember AppBuilder Help at Help | Help Contents | Creating custom clusters

app/framework: All of the Ember application framework code is located in app/framework. Major portions of the code have been broken out into their own directories.

app/framework/cli: Code related to the application framework's implementation of the Command Line Interface.

Core code for the CLI is included in app/util/serial/command-interpreter2.c. The CLI includes data type checking and command usage feedback among other things. As a result:

1. All commands require ALL arguments associated with that command. If an argument is missing, the CLI will provide user feedback as to the particular commands usage.

2. Arguments passed with the CLI must be in one of the following formats:
   <int>: 123(decimal) or 0x1ABC(hex)
   <string>: "foo"(string) or {0A 1B 2C}(array of bytes)

app/builder: The output location for all generated files from the Ember AppBuilder

When you generate an application from Ember AppBuilder, it puts the generated files into this directory under a directory of the same name as the device name within the Ember AppBuilder configuration. For instance, if your device is named MyLightSwitch, files are generated into app/builder/MyLightSwitch/. The same is true when a sample application is opened in Ember AppBuilder. Ember AppBuilder automatically copies the sample application files into the associated directory within app/builder.

app/framework/include: All of the external APIs for the Ember application framework

This directory mirrors the use of the include directory in the stack. It is intended to be the single location for all externally facing application interfaces.

app/framework/plugin: All Silicon Labs-created ZCL cluster code

This directory contains all of the cluster code created by the Silicon Labs team for handling cluster commands. This code optionally can be included in an application by selecting the plugin from Ember AppBuilder's plugin pane. If you choose not to include a plugin, you are responsible for implementing the callbacks for all of the required cluster commands.

app/framework/sample-apps: All sample applications which use the application framework

These sample applications may be opened within Ember AppBuilder. Ember AppBuilder requests a new application name for the given sample application and copies the sample application into a directory of the same name within app/builder.

app/framework/security: All utility code related to ZigBee Security

Code related to key establishment is located in app/framework/cluster.

app/framework/util: The application's mains, message processing, and any other utility code used by the Ember application framework

This directory contains the guts of the Ember application framework. Attribute storage files that manage attributes for multiple endpoint support are included in this directory. In addition, the API used for accessing, reading, and writing attributes is included in the file attribute-table.h, and attribute-storage.h.

4 Generated Application Configuration Files

Version 1 of the Ember application framework used a single header file to configure the Ember application framework, set up the attribute table, and turn on and off portions of the code through preprocessor directives. Version 2 uses the same preprocessor directives to configure the code to be included and excluded from the framework. In addition to the main app header file, Ember AppBuilder also generates an "endpoint configuration" header file with the suffix endpoint_configuration.h.

<DeviceName>_endpoint_configuration.h

The generated file that configures the Ember application framework's static data structures. This allows attribute metadata to be shared across endpoints, and each endpoint to have its own space for attribute storage. The #defines in the endpoint_configuration.h file are used by the app/framework/util/attribute-storage.c file to configure all of the application's attribute-related data.

The file must be re-generated each time you modify your application configuration in Ember AppBuilder. Silicon Labs recommends that you do not edit the endpoint_configuration.h file by hand as each of the macro definitions in the file has a complex relationship.

The role of the endpoint configuration file is described in more detail in section 9.1, ZCL Attribute Configuration.

<DeviceName>.h

The main header file for your application. It includes all of the #defines that turn on the features you require within the framework.

<DeviceName>_callbacks.c

A generated stub callback file containing default implementations of all callbacks you have selected to include in your project. This is where your code goes. You are not restricted to using this one file for your code. You can include other files provided you add them to your generated project file so that they can be found by the compiler.

<DeviceName>_board.h

The generated board file for your chosen platform. This file assumes that you are using one of the Ember development boards. It is configured according to the selections you have made in the HAL configuration tab.

Note: The board header file contains many options for configuring GPIO differently for the needs of your board. Silicon Labs recommends you review the generated <DeviceName>_board.h file and make changes as needed for your target hardware.

<DeviceName>_tokens.h

If you are including any attributes in tokens (persistent memory) for a platform that supports tokens, this file is generated by Ember AppBuilder to configure your token storage.

<DeviceName>.ewp, eww, .xip, .xiw, .mak

Generated project files for your application. Ember AppBuilder only generates the project files that match the platform you have chosen. These files may be loaded into your IDE and edited to build out the rest of your project.

4.1 Application Framework Files

As of EmberZNet PRO 4.6, the Ember application framework and the Ember AppBuilder have been modified to generate all of the static Ember application framework files into the application build directory. In prior releases of the EmberZNet PRO stack, a single version of these files was generated into the app/framework/gen directory. As of EmberZNet PRO 4.6, this directory has been removed. Instead all of the files are generated into app/builder/<application name> directory. This was done in order to support multiple specification versions from a single stack. The generated files are no longer static. Their contents change based on the ZigBee specification versions chosen by the user in the Stack Tab.

The number of files generated varies based on what plugins are supported and what is required for those plugins. The Ember application framework files that are generated include but are not limited to the following:

af-structs.h: Definitions of structures used by the Ember application framework for the parsing of data sent over the air.

att-storage.h: Defines used in the attribute storage mechanism within the Ember application framework.

attribute-id.h: All attribute ids defined by the ZigBee Cluster Library specifications for all profiles loaded into the Ember application framework.

attribute-size.h: Size in bytes for attribute types used in the ZigBee Cluster Library specification.

attribute-type.h: Defines to represent over the air values for data types used in the ZigBee Cluster Library specification.

call-command-handler.c: Command handling code for all non-general commands received over the air. This generated code marshals cluster commands from their over the air format off to the callback interface. It also handles the commands if no callbacks are implemented for them.

call-command-handler.h: Header file for the call-command-handler c code. This file provides definitions for all of the functions implemented in call-command-handler.c.

callback-stub.c: Provides stubs for custom callbacks implemented by the Ember application framework. The callback stubs are only compiled in if they are not separately defined by the customer's application.

callback.h: Provides definitions for ALL callbacks that can possibly be implemented within the Ember application framework or the users application. This defines the ENTIRE callback interface which is the main interface used by the Ember application framework when communicating with the user application.

cli.h: This file is used only by the documentation engine doxygen to document general application framework cli commands it has no other purpose.

client-command-macro.h: Macros that are provided as a convenience as part of the Ember application framework interface in the filling of packet buffers that will be sent over the air. Each command supported by the ZigBee Cluster Library as configured in the user's Application Configuration is represented here with a macro that will make the appropriate calls into the Ember application framework to fill a packet buffer to send that command over the air.

cluster-id.h: Defines provided for all cluster ids for loaded into the Ember application framework from the ZigBee Cluster Library.

command-id.h: Defines provided for all command ids loaded into the Ember application framework from the ZigBee Cluster Library.

debug-printing-test.h: Defines used to turn on debug printing within the Ember application framework.

debug-printing.h: Macros used for debug printing within the Ember application framework.

enums.h: Provides definitions for all ZigBee Cluster Library related enums used in the Ember application framework.

print-cluster.h: Defines used to turn printing on on a per cluster basis within the Ember application framework.

stack-handler-stub.c: Stubs for all stack handlers which are available to be overridden within the Ember application framework.

stack-handlers.h: Defines for all stack handler functions which are available to be overridden by the user application.

<plugin>-callbacks.h: Declarations for all plugin callback functions. These files are generated depending on which plugins are selected for your application.

5 The Application Framework API

The Ember application framework's API is provided in app/framework/include/af.h. This interface file is consistent with the way the EmberZNet PRO API is exposed by the stack. HTML and PDF versions of document 120-3023-000, the *Application Framework API Reference*, are provided with your installation.

Many of the functions in the Ember application framework include a passed one-byte endpointId. This is particularly true for functions like cluster initialization, cluster ticks, and attribute management. For instance, the function zclUtilReadAttribute is located in app/framework/util/attribute-table.c, and the signature of the function takes the endpointId as its first argument.

Some examples of the Ember application framework include:

```
boolean emberAfContainsCluster(int8u endpoint, EmberAfClusterId clusterId);
boolean emberAfContainsServer(int8u endpoint,  EmberAfClusterId clusterId);
boolean emberAfContainsClient(int8u endpoint, EmberAfClusterId clusterId);
```

All of the Ember application framework APIs intended to be used by the customer application include the "emberAf" prefix.

APIs for getting information about endpoints and attributes are included in app/framework/util/attribute-storage.h. For instance, to determine if an endpoint contains a certain attribute, use the function

emberAfContainsAttribute(int8u endpoint, ClusterId clusterId, AttributeId attributeId). It returns a Boolean indicating if the requested attribute and cluster are implemented on the specific endpoint.

Note: The read and write attribute needs an endpoint. If you do not include one, the compiler returns a warning that the function is declared implicitly, but not a compiler error. Therefore, pay attention to warnings.

6 Application Framework Callback Interface

The Ember application framework callbacks are intended to be used as a means to remove all customer code from the Ember application framework. If any of your application code needs to be put into the Ember application framework, Silicon Labs views this as a bug with the Ember application framework, because it means that a callback that would satisfy your application requirement is missing. In this case, please open a ticket on the support portal at www.silabs.com/zigbee-support.

Generally, when a callback is called the Ember application framework is giving the application code a first crack at some incoming message or requesting some piece of application data. Within the callback API, some callbacks return a Boolean value indicating that the message has been handled and no further processing should be done. If you are doing something that conflicts with the Ember application framework's handling of a particular message, return TRUE to indicate that the message was complete. This ensures that the Ember application framework does not interfere with your handling of the message.

6.1 Callback Generation

Ember AppBuilder has the ability to generate a stub callback file for you. By default, Ember AppBuilder chooses not to generate the callback stub file if it finds that the file already exists in the generation directory. You must specifically tell the application to overwrite an existing file.

When you regenerate files in the future, Ember AppBuilder protects your generated callbacks file from being overwritten by asking if you want to overwrite it. By default, Ember AppBuilder will not overwrite any previously created callbacks file. If you choose to overwrite the file, Ember AppBuilder backs up the previous version to the file <appname>_callbacks.bak.

Note: You can implement your callbacks wherever you want; they do not need to be implemented in the generated callbacks file. However if you implement them in a different location, clear them out of the generated callback file so that your linker won't complain about duplicate definitions for the callback functions.

6.2 Non Cluster-Related Callbacks

The callback interface is divided up into sections within the Ember AppBuilder GUI for ease of use. The first section, Non Cluster-Related Callbacks, is made up of callbacks that are described in the callbacks.xml document located at tool/appbuilder/callbacks.xml. These callbacks have been manually inserted into the Ember application framework code in locations where customers have indicated that they wish to receive information about the function of the Ember application framework.

All global commands fall into this category. The Ember application framework contains handling code for global commands. If any global command callback returns TRUE, this indicates that the command has been handled by the application and no further command handling should take place. If the callback returns FALSE, then the Ember application framework continues to process the command normally.

Example

The pre-command received callback (`emberAfPreCommandReceivedCallback(EmberAfClusterCommand* cmd, boolean isInterpan`) is called after a ZCL command has been received but has not yet been processed by the Ember application framework's command handling code. The command is parsed into a useful struct `EmberAfClusterCommand`, which provides an easy way to access relevant data about the command including its `EmberApsFrame`, message type, source, buffer, length, and any relevant flags for the command. This callback also returns a Boolean value indicating if the command has been handled. If the callback returns TRUE, then it is assumed that the command has been handled by the application and no further action is taken.

6.3 Cluster-Specific Command Handling Callbacks

The cluster-related callbacks are generated by the Ember application framework to allow receipt of a pre-parsed command coming over the air. Generally a one-to-one relationship exists between ZCL commands and the cluster-specific callbacks.

The cluster-specific command callbacks all return a Boolean value. This return value allows you to short-circuit command handling included in the application framework. If you implement a cluster-specific command callback and it returns a value of TRUE to the Ember application framework, the framework assumes that the command has been handled outside the framework and that any required command or default response has been sent. If the cluster-specific command returns FALSE, the framework assumes that the application code did not understand the command and sends a default response with a status of 'unsupported cluster command'.

6.3.1 Command Callback Context

All command-related callbacks are called from within the context of the emberIncomingMessageHandler. This means that Ember APIs that are available to the application within that context are available within the command handling callbacks as well. These APIs are listed in the stack API file located at stack/include/message.h. The stack APIs that are available in the command callbacks are listed in the stack message header located at stack/include/message.h and include:

```
emberGetLastHopLqi()
emberGetLastHopRssi()
emberGetSender()
emberGetSenderEui64()
emberGetBindingIndex()
emberSendReply() (for incoming APS retried unicasts only)
emberSetReplyBinding()
emberNoteSendersBinding()
```

6.3.2 Array Handling in Command Callbacks

Any ZigBee message that contains an array of arguments is passed as an int8u* pointer to the beginning of the array. This is done even when the framework knows that the arguments in the array may be of another type, such as an int16u or int32u, because of byte alignment issues on the various processors on which the framework may run. Developers implementing the callback must parse the array and cast its elements appropriately for their hardware.

6.3.3 Global Command Callbacks

ZigBee global commands are also covered in the Ember application framework callback interface. These callbacks can be used to receive responses to global commands. For instance, if your device sends a global read attribute command to another device, it can process the command response by implementing the emberAfReadAttributesResponseCallback.

6.4 Callback Flow

Figure 6 1 shows how a message received by the application framework's implementation of emberIncomingMessageHandler is processed and flows through the framework code and out to the application implemented callbacks.

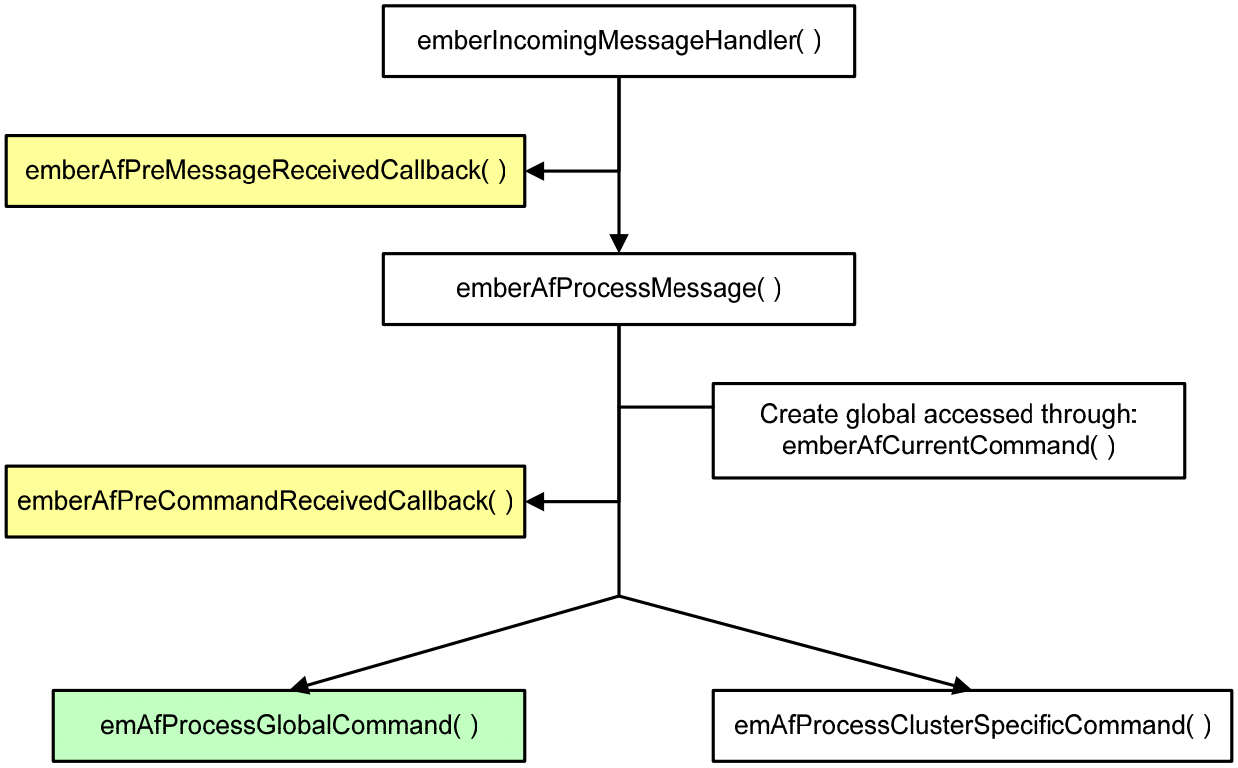

Figure 2. Incoming Message Flow

Once the incoming message is determined to be an incoming global command, it is passed off to the global command handling for processing, as shown in Figure 3.

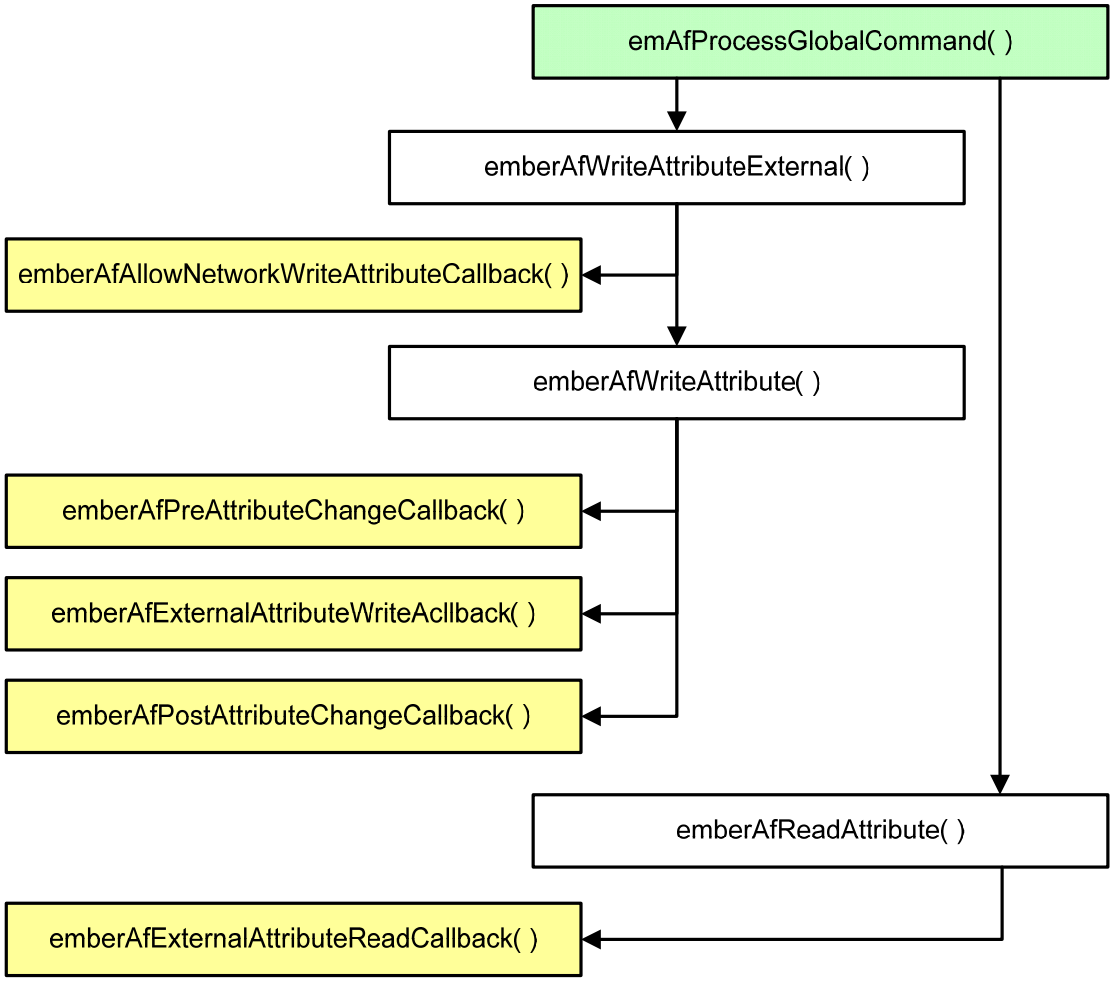

Figure 3. Global Command Handling

Otherwise, if it is found to be a cluster specific command, it is passed off to the cluster-specific command processing, as shown in Figure 4.

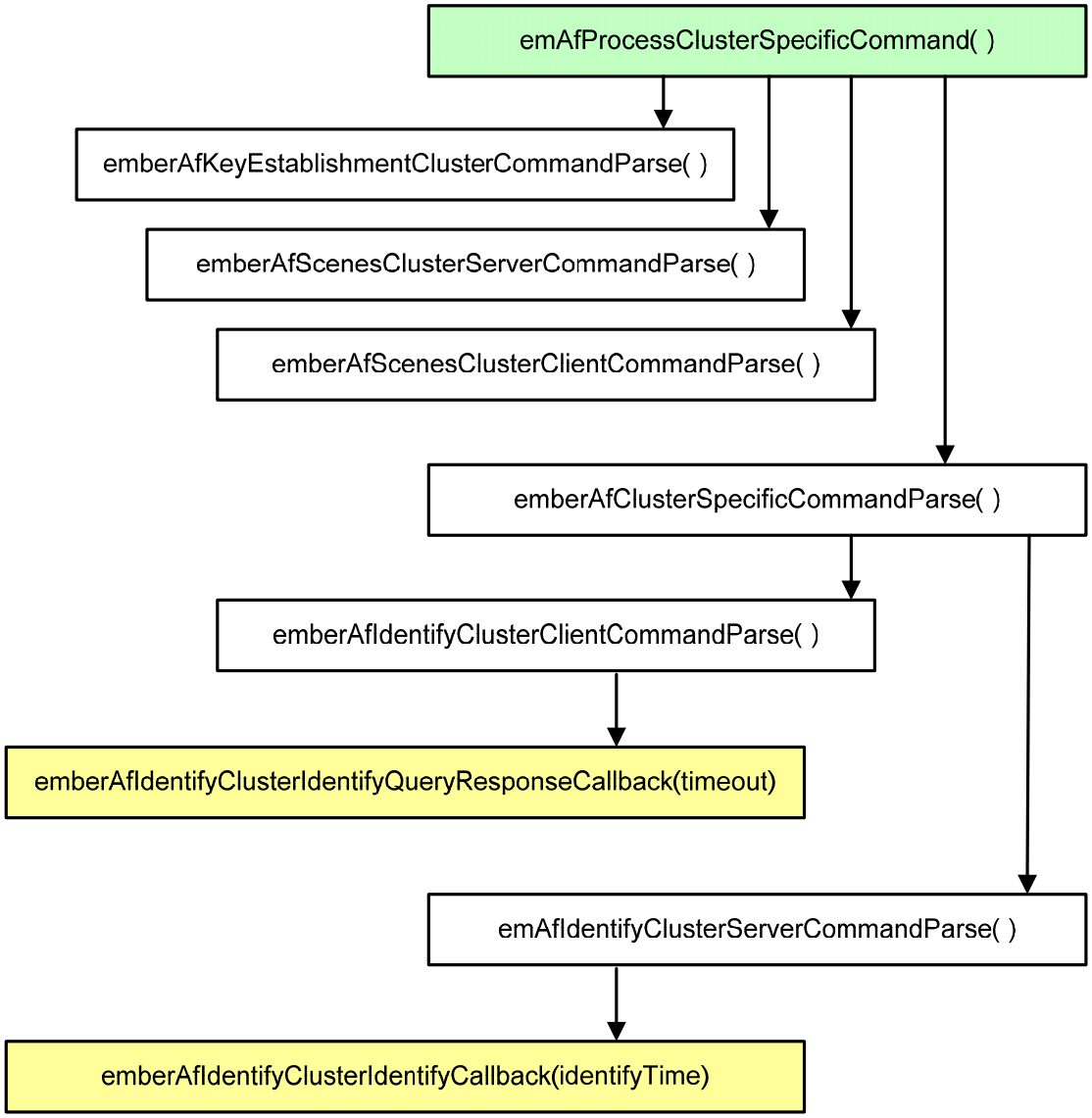

Figure 4. Cluster-Specific Command Processing

6.5 Callback Reference

Note The callback reference that was included in this document has been removed in favor of the reference provided in document 120-3023-000, the *Ember Application Framework API Reference*. The reference included in that document is generated directly from the header files and as a result is more up-to-date than the reference provided in this document.

7 Time Handling

The Ember application framework provides a single API for accessing the current time on the system (`int32u emberAfGetCurrentTime()`), which is described in app/framework/include/af.h. This section describes how the function is implemented in app/framework/util/util.c.:

If the ZCL time cluster server is implemented on the system, then this function retrieves the time from the server through the function call (`int32u emberAfTimeClusterServerGetCurrentTime()`), in which case the time is read from the time cluster server's time attribute and returned. If the time cluster server is not implemented, then `emberAfGetCurrentTime` calls `emberAfGetCurrentTimeCallback`.

If your device needs to know the current time but does not implement the time cluster server plugin, it is responsible for maintaining its own time somewhere on the system and returning that time through the `emberAfGetCurrentTimeCallback` when it is requested. This is especially important for SE devices that do not implement the time cluster server, like an in-premise display (IPD). Essentially the IPD is on its own when it comes to time management. It would be outside the specification (as currently interpreted) for a non-Energy Service Portal to implement the time cluster server. Therefore, the IPD must maintain its own knowledge of time and provide it to the framework when requested through the `emberAfGetCurrentTimeCallback`.

If your application includes the time cluster server, the time cluster server code always tries to initialize and update the time server's time attribute through the `emberAfGetCurrentTimeCallback`. If the `emberAfGetCurrentTimeCallback` returns 0, then the time cluster server increments the stored attribute once per second. Thus you can use the time cluster server to store and maintain real time on the system without implementing the `emberAfGetCurrentTimeCallback`, if the actual time value can be synced from another device on the system and written into the time server's time attribute. For more information on how time is handled by the bundled implementation of the time cluster server see app/framework/plugin/time-server/time-server.c.

The Ember application framework includes a time client plugin that allows time clients to sync and keep track of the current UTC time without having to implement a time cluster server. If your device is something other than an ESP, it should implement the time cluster client and use the included time client plugin to keep track of time.

8 Events

The Ember application framework and its associated cluster code use the Ember event mechanism to schedule events on both the SoC and the host. Use of the Ember event mechanism saves code and RAM, and works better with sleepy devices.

At a high level, the event mechanism provides a central location where all periodic actions taken by the device can be activated and deactivated based on either some user input, an over-the-air command or device initialization. The event mechanism is superior to the constant tick mechanism it replaces because it allows the Ember application framework to know precisely when the next action is going to occur on the device. This is extremely important for sleeping devices that need to know exactly when they must wake up to take some action - or more importantly that they cannot go to sleep because some event is in progress. The Ember application framework has two types of events: custom events and cluster events. Custom events are created by the Ember application framework user and can be used for any purpose within the application. Cluster events are specifically related to the cluster implementations in the Ember application framework's plugins.

8.1 Creating a Custom Event

The Ember application framework uses the Ember standard event mechanism to control and run "custom" application events within the Ember application framework. The stack's event mechanism is documented in the event.h header file located at stack/include/event.h.

The Ember application framework and Ember AppBuilder provide a helpful interface for creating and adding custom events to your application. To create a custom event in Ember AppBuilder, open the “Includes” tab in your Ember AppBuilder configuration file. In the “Custom Events” section click on the “New” button to create a custom event. This adds an event to the list of events that will be run by the Ember application framework, as well as stubs for your custom event to the “callbacks” file generated by Ember AppBuilder.

8.1.1 Event Function and Event Control

A custom event consists of two parts: The event function, called when the event fires, and the EmberEventControl struct, which is used to schedule the event. The framework’s event mechanism must know about each of these items so that it can both keep track of when the next event will occur for the purposes of sleeping and also so that it knows what function to call when the event fires. Further documentation on creating an event is provided in the event.h header file located at stack/include/event.h.

8.1.2 Custom Event Example

The SeMeterGasSleepy sample application uses a custom event to manage its state. The event consists of two parts, the EmberEventControl struct called sleepyMeterEventControl and the event function, which is called each time the event fires. The event function is called the sleepyMeterEvent. The event and event controls are included in the configuration file shipped with the sample application. Documentation for the sleepy meter and the sleepy meter code are provided in the README.txt file and the SeMeterGasSleepy_callbacks.c file located in the sample application directory at app/framework/sample-apps/se-meter-gas-sleepy/.

8.2 How Cluster Events Are Created

Every cluster includes a server and a client “tick” callback. Ember AppBuilder generates an event table with a single event for each cluster server or client on each endpoint. The actual event table is generated into the <DeviceName>_endpoint_config.h header, which is included and used in the Ember application framework’s event code in app/framework/util/af-event.c.

Note: The event table is created at compile time and is static. Thus, events cannot be randomly added or removed from the event table at runtime. The event table entry must be present, and then the code can manage its schedule so that it is either active and waiting to be called, or deactivated and waiting to be activated and scheduled.

8.3 How Cluster Events Are Scheduled

The plugin or application code can manage cluster-related events in the event table by using the Ember application framework’s event management API. This API consists of two functions, `emberAfScheduleClusterTick` and `emberAfDeactivateClusterTick`.

8.3.1 emberAfScheduleClusterTick

`emberAfScheduleClusterTick` uses the endpoint, cluster id, and client/server identity to find the associated event in the event table. The event table entry is generated by Ember AppBuilder into `<DeviceName>_endpoint_config.h`. If it cannot find the event table entry, `emberAfScheduleClusterTick` returns the EmberStatus EMBER_BAD_ARGUMENT to the caller. If it finds the event table entry, then it schedules the event to take place in the number of milliseconds requested by the caller, and it returns EMBER_SUCCESS.

```
EmberStatus emberAfScheduleClusterTick( int8u endpoint,
                                        int16u clusterId,
                                        boolean isClient,
                                        int32u timeMs,
                                        EmberAfEventSleepControl sleepControl);
```

The EmberAfEventSleepControl argument allows the caller to indicate what the device may do while the event is active in the event table. This value is only relevant for sleepy devices; it has no effect for devices that do not go to sleep. The possible values for EmberAfEventSleepControl are enumerated in app/framework/include/af-types.h, as follows:

- EMBER_AF_OK_TO_HIBERNATE means that the application may go into prolonged deep sleep until the event needs to be called. Use this sleep control value if the scheduling code does not care what the device does up to the point when the event is called.
- EMBER_AF_OK_TO_NAP means that the device should sleep for the nap period and should wake up to poll between naps until the event is called. Use this sleep control value if the scheduling code wants the device to poll periodically until the event is called. This is particularly useful if the scheduled event is a timeout waiting for some reply from another device on the network. If the event is a timeout, you don't want the device to go into hibernation until the timeout is called, because it will never hear the message it is waiting for, thereby guaranteeing that the timeout will be called.
- EMBER_AF_STAY_AWAKE means that the device should not sleep at all but should stay awake until the event is called. Use this event if you are scheduling a very frequent event and don't want the device to nap for a very short period of time since the device will poll each time it wakes up. If the device is held out of sleep entirely, it will poll once per second.

8.3.2 emberAfDeactivateClusterTick

The deactivation function is used to turn off an event. This function should be called when the scheduled event is called to ensure that the event code does not continue to call the event. It may also be called before the event is called if the event is no longer necessary.

Note: In the Ember application framework `emberAfDeactivateClusterTick` is automatically called before the event fires to ensure that the event will not continue to be called on every tick. You can see the call to `emberAfDeactivateClusterTick` in the generated event table output from Ember AppBuilder as of version 2.1.50.

DeactivateClusterTick is similar to ScheduleClusterTick in that it takes most of the same arguments, since it also has to locate the clusterTick in the event table before shutting it off.

```
EmberStatus emberAfDeactivateClusterTick(int8u endpoint,
                                         int16u clusterId,
                                         boolean isClient);
```

9 Attribute Management

9.1 ZCL Attribute Configuration

In the Ember application framework , attribute storage is managed by two .c files (app/framework/util attribute-storage.c and attribute-table.c) as well as a single header file (I<appname>_endpoint_config.h), which Ember AppBuilder generates from the application configuration. The endpoint configuration header file sets up the attribute metadata and the actual attribute storage.

You have several options for attribute storage:

- External Attributes
- Persistent Memory Storage
- Singleton
- Attribute Bounding
- Attribute Reporting

9.1.1 Attribute Storage Endianness

All attributes that are not a ZCL string type are expected to be stored with the same endianness as the platform on which the application is being run. For ARM® Cortex□-M3 based chips like the EM35x series and host chips like the AVR 128 or the Stm32f103, this means that attributes with a non string type are expected to be stored with the Least Significant Byte first (LSB, Little Endian). In the case of the Xap2b based chips like the EM250 this means that they are expected to be stored with the Most Significant Byte first (MSB, Big Endian).

9.1.2 Implications of Attribute Storage Endianness

The ZigBee protocol demands that all values that are not a string or byte array type be sent over the air in a Little Endian or LSB format. The implication of this for the 35x and AVR 128 is that no byte swapping needs to be done with attributes when they are pulled from attribute storage and sent over the air. Conversely, when the Ember application framework is run on the EM250 it will perform byte swapping on integer type attributes before they are sent over the air so that they are sent in the LSB format.

Section 9.1.1 above says that attributes are expected to be stored in the proper format because no byte swapping is done on local writes into the attribute table from native types or from byte arrays. Therefore it is up to the user to ensure that byte arrays which represent ZigBee integer types but do not map directly to a native type like the int16u or int32u are represented in the byte order of the application platform.

If you are writing an application that may be run on several platforms with different endianness, you may check the endianness of the platform by using the #define BIGENDIAN_CPU provided by the hal shipped with the Ember stack.

Example: Consider the simple-meter-server plugin's test code located at app/framework/plugin/simple-metering-server/simple-metering-test.c. This test code pulls the simple metering daily summation attribute from the attribute table, updates it, and puts it back into the attribute table. Unfortunately, the daily summation is a ZigBee 48-bit unsigned integer, which is not a native data type.

The Ember HAL for the 35x family of processors has no native data type like an int48u into which the daily summation attribute can be read and simply manipulated. As a result, the attribute must be read into a byte array and the byte array must be manipulated before it is written back into the attribute table. During this manipulation it is important for the developer to remember that on the EM357 the attribute is stored LSB, so the manipulation must be done LSB. Otherwise the value will be stored and sent over the air in the wrong format when it is read by another device on the network.

The Ember application framework is written to run on any platform the user desires. As a result, the Ember application framework code first checks the CPU's endianness before doing any manipulation to the daily summation value to ensure that it maintains the proper LSB format on the EM357 and MSB format on the EM250.

Note: While the EM260 is a big-endian CPU like the EM250, the endianness of the CPU does not matter for attribute storage since all attributes are stored on the host processor. In an NCP + Host design, it is the endianness of the host that counts for attribute storage.

9.1.3 External Attributes (E)

You may wish to store the values for some attributes in a location external to the Ember application framework. This type of storage makes the most sense for attributes that must be read from the hardware each time they are requested. In a case like this, no real reason exists to store a copy of the attribute in some wasted RAM space within the Ember application framework.

Mark an attribute as externally located by clicking on the “E” checkbox next to the attribute in the Ember AppBuilder GUI. The attribute’s metadata will be tagged to indicate that the Ember application framework should not reserve memory for the storage of that attribute. Instead, when that attribute is to be read or written, the Ember application framework accesses it by calling emberAfExternalAttributeReadCallback and emberAfExternalAttributeWriteCallback.

Note: Once you designate a single attribute as “External” these two callbacks are automatically included in your generated callback.c file.

The application is expected to respond to the request immediately. No state machine is currently associated with accessing external attributes that would be able to, for example, start a read and then callback again in a minute to see how the data read is going.

Any attribute that cannot be returned or updated in a timely manner is not currently a candidate for externalization. For attributes of this type, Silicon Labs suggests that you include Ember application framework storage and update the value in the Ember application framework on a specific interval within the emberAfMainTickCallback.

9.1.4 Persistent Memory Storage (F)

The Ember System-on-Chip (SoC) chips, like the EM250 and EM35x, can store attributes in persistent memory (SIMEEPROM). In these cases, mark the attribute for persistent storage by clicking on the “F” checkbox next to the attribute in the Ember AppBuilder GUI. This automatically adds the necessary header file code to the generated <appname>_tokens.h file and marks the attribute as persisted in flash within the attribute’s metadata.

Because each host chip has its own way of storing persistent data, the Ember application framework and Ember AppBuilder do not have a way of persisting attributes on the host. However, you can mark any attribute you wish to persist as ‘External’ and then handle the data persistence yourself within emberAfExternalAttributeReadCallback and emberAfExternalAttributeWriteCallback.

9.1.5 Singleton (S)

While ZCL clusters and attributes can be spread across multiple endpoints, it does not make sense to have multiple instances of many of these attributes. For instance, the Basic Cluster may be implemented on three different endpoints, but it doesn’t make sense to store three versions of the mandatory ‘ZCL Version’ attribute, since each endpoint will likely have the same version. Mark attributes like this by clicking on the checkbox marked “S” next to the attribute in Ember AppBuilder. As a convenience, the Ember application framework provides a default ‘Singleton’ modifier for many of the obvious cases. This default modifier can be overridden if you choose.

Attributes marked as singleton are stored in a special singleton storage area in memory. A read or write to any endpoint for one of these attributes resolves to an access of the same location in memory.

9.1.6 Attribute Bounding (B)

Attributes which contain min and max values defined by the ZigBee ZCL specification can be bounded within the Ember application framework. When an attribute is bounded, the min and max values defined by the ZCL specification are included in the generated <appname>_endpoint_config.h file. When the application attempts to write one of these attributes, the attribute write succeeds only if its value falls within the bounds defined by the ZCL specification.

9.2 Interacting with ZCL Attributes

The Ember application framework attributes table exposes several APIs that help you do things like read, write, and verify that certain attributes are included on a given endpoint. The prototypes for functions used to interact with the attribute tables are conveniently located in app/framework/include/af.h. The API includes:

`emberAfLocateAttributeMetadata:` Retrieves the metadata for a given attribute

Use this function to determine if the attribute exists or is implemented on a given endpoint. You can use the emberAfAttributeMetadata pointer returned to access more information about the attribute in question including its type, size, defaultValue and any internal settings for the attribute contained in its mask.

```
EmberAfAttributeMetadata *emberAfLocateAttributeMetadata(int8u endpoint,
EmberAfClusterId cluster, EmberAfAttributeId attribute);
```

The Ember application framework stores metadata for all of the attributes that it contains in CONST memory. It does this for all attributes, including those that may have values stored externally or singletons.

9.2.1 ZCL String Attributes

The String data type is a special case in the ZCL. All strings are MSB with the first byte being the length byte for the string. There is no null terminator or similar concept in the ZCL. Therefore a 5 byte string is actually 6 bytes long, with the first byte indicating the length of the proceeding string. For example, “05 68 65 6C 6C 6F” is a ZCL string that says “hello.”

10 Command Handling and Generation

10.1 Sending Commands and Command Responses

The Ember application framework API includes many useful macros for sending and responding to ZCL commands. All of the macros are defined in the file client-command-macro.h. This file is generated for each project. For example, after building project STM32, the file can be found in <install path>/app/builder/STM32/client-command-macro.h.

To send a command, do the following:

Sending a command:

1. Construct a command using a fill macro from client-command-macro.h file:

   For example:
   `emberAfFillCommandIdentifyClusterIdentify(identifyTime);`
   identifyTime is an int16u defined in the spec as the number of seconds the device should continue to identify itself.
   This macro fills the command buffer with the appropriate values.
2. Retrieve a pointer to the command EmberApsFrame and populate it with the appropriate source and destination endpoints for your command. Other values in the ApsFrame such as sequence number are handled by the framework, so you don’t need to worry about them.
3. Once the command has been constructed, the command can be sent as a unicast, multicast, or broadcast using one of the following functions

   ```
   EmberStatus emberAfSendCommandMulticast(int16u multicastId);
   EmberStatus emberAfSendCommandUnicast(EmberOutgoingMessageType type,
   int16u indexOrDestination);
   EmberStatus emberAfSendCommandBroadcast(int16u destination);
   ```

Sending a response to an incoming command:

Use a similar mechanism to send a response to an incoming command.

1. Fill the response command buffer using the command response macros included in app/framework/gen/client-command-macro.h such as:

   ```
   emberAfFillCommandIdentifyClusterIdentifyQueryResponse(timeout)
   ```

   Timeout is an int16u representing the number of seconds the device will continue to identify itself.
2. You don't need to worry about the endpoints set in the response EmberApsFrame since these are handled by the framework.
3. Send the response command by calling `emberAfSendResponse`().

10.2 ZCL Command Processing

When the Ember application framework receives a ZCL command, it is passed off for command processing inside the utility function emberAfProcessMessage, located within app/framework/util/util.c. The process message function parses the command and populates a local struct of the type EmberAfClusterCommand. Once this struct is populated, it is assigned to the global pointer emAfCurrentCommand so that it is available to every function called during command processing.

EmberAfProcessMessage first calls emberAfPreCommandReceivedCallback to give the application a chance to handle the command. If the command is a global command, it is passed to process-global-message.c for processing; otherwise, it is passed to process-cluster-message.c for processing.

Note: For more information on command processing flow, please see the message flow charts included in Chapter 6 regarding the callback interfaces.

10.2.1 app/framework/util/process-global-message.c

Process-global-message.c handles all global commands, such as reading and writing attributes. Global commands do not currently have associated command callbacks the way cluster-specific commands do.

10.2.2 app/framework/util/process-cluster-message.c

Process-cluster-message.c handles all cluster-specific commands. Most cluster-specific commands are in turn passed to the generated file call-command-handler.c located at app/framework/gen/call-command-handler.c. This generated file parses the command's parameters and optionally calls the associated cluster-specific callback.

The generated file call-command-handler.c currently does not handle key establishment. Command handling was deemed too complex for the current command handler generator. Commands for key establishment are passed directly to the cluster code for processing in app/framework/cluster/key-establishment.c.

Note: Since the cluster-specific command callbacks are called within the command handling context, all of the metadata associated with any command handled in one of these callbacks is available from the global pointer emAfCurrentCommand.

Always access the global pointer emAfCurrentCommand by using the convenience macro provided in app/framework/include/af.h called emberAfCurrentCommand().

10.3 Sending a Default Response

The Ember application framework does not automatically send a default response for command callbacks implemented by the application. In order to improve system reliability and flexibility, we have handed all the default

response handling over to the application. This means that, while you now have complete control over sending default responses for commands that you handle, you also are responsible for sending default responses for all those commands. A default response must be sent for any unicast message that does not have a specific response and is not itself a default response. For more information on when default response should and should not be sent, please refer to the ZigBee documentation.

The Ember created plugins handle sending default responses for all of the commands that they handle. Any commands that the plugins do not handle automatically return EMBER_ZCL_STATUS_UNSUP_CLUSTER_COMMAND, or something like it. Your application needs to do the same for all of the commands that it handles that do not themselves have a specific command response.

We have created a default response API to make this is simple as possible. The emberAfSendDefaultResponse command takes two arguments: the current command, and the status byte. The current command can be retrieved from the Ember application framework using emberAfCurrentCommand(). The ZCL status bytes used for default response are enumerated in app/framework/gen/enum.h.

```
void emberAfSendDefaultResponse(EmberAfClusterCommand *cmd, EmberAfStatus status);
```

A typical use of this function looks like:

```
emberAfSendDefaultResponse( emberAfCurrentCommand(), EMBER_ZCL_STATUS_SUCCESS );
```

11 The Command Line Interface (CLI)

The Ember application framework includes a command line interface (CLI) that implements many common commands and cluster-specific commands. For instance, commands related to common functionality, like network formation and attribute read and write, are implemented by the CLI.

The Ember application framework CLI can take integer arguments as both decimal and hexadecimal notation. If an argument includes the 0x prefix, it is assumed to be hexadecimal, otherwise decimal. In addition, arrays of integers may be passed within curly braces, and strings may be passed inside quotations.

11.1 Extending the Command Line Interface

The process for extending the command line interface has changed. We have deprecated the use of emberAfCustomCommandLineCallback in favor of allowing you to define as many command line options as you like using the existing command line architecture. The new process for extending the command line interface allows you to add an entire array of commands directly into the command processor. The process is as follows:

1. In the “Includes” tab in Ember AppBuilder, add a macro entitled EMBER_AF_ENABLE_CUSTOM_COMMANDS. This enables the inclusion of a command array called emberAfCustomCommands in the file /app/framework/cli/custom-cli.h that is extern’d in that file. You must now provide a definition for the emberAfCustomCommands array to satisfy the requirements of the linker at compile time.
2. Define an array of type EmberCommandEntry called emberAfCustomCommands in your application. An example of this is provided in the se tunneling sample application located at /app/framework/sample-apps/se-tunneling/SeTunneling_callbacks.c
3. The example provided in the SeTunneling sample application creates a single custom command called “custom start.” The SeTunneling example does not take any arguments. The example below adds two commands, “form” and “join.” These commands each take four arguments described in the short hand form “uvsh.” The short hand used to describe arguments to a command is described in the next section.

```
// The table of network commands.
EmberCommandEntry networkCommands[] = {
```

```
  { "form",       formCommand, "uvsh" },
  { "join",       joinCommand, "uvsh" },
  ...
  { NULL }
};
```

4. All of the commands that you define are of the type EmberCommandEntry. The type EmberCommandEntry is documented in the stacks API reference. Basically they are of the form “<string> command”, <function>, “<string> args”, where the arguments are a string of characters indicating underlying types of the passed arguments. The definition for EmberCommandEntry from /app/util/serial/command-interpreter2.h is included below.

```
typedef PGM struct {
#endif
  /** Use letters, digits, and underscores, '_', for the command name.
   * Command names are case-sensitive.
   */
  PGM_P name;
  /** A reference to a function in the application that implements the
   *  command.
   *  If this entry refers to a nested command, then action field
   *  has to be set to NULL.
   */
  CommandAction action;
  /**
   * In case of normal (non-nested) commands, argumentTypes is a
   * string that specifies the number and types of arguments the
   *  command accepts.  The argument specifiers are:
   *  - u:   one-byte unsigned integer.
   *  - v:   two-byte unsigned integer
   *  - w:   four-byte unsigned integer
   *  - s:   one-byte signed integer
   *  - b:   string.  The argument can be entered in ascii by using
   *         quotes, for example: "foo".  Or it may be entered
   *         in hex by using curly braces, for example: { 08 A1 f2 }.
   *         There must be an even number of hex digits, and spaces
   *         are ignored.
   *  - *:   zero or more of the previous type.
   *         If used, this must be the last specifier.
   *  - ?:   Unknown number of arguments. If used this must be the only
   *         character. This means, that command interpreter will not
   *         perform any validation of arguments, and will call the
   *         action directly, trusting it that it will handle with
   *         whatever arguments are passed in.
   *  Integer arguments can be either decimal or hexidecimal.
   *  A 0x prefix indicates a hexidecimal integer.  Example: 0x3ed.
   *
   *  In case of a nested command (action is NULL), then this field
   *  contains a pointer to the nested EmberCommandEntry array.
   */
  PGM_P argumentTypes;
  /** A description of the command.
   */
  PGM_P description;
} EmberCommandEntry;
```

11.2 CLI Examples

11.2.1 Example 1: Creating a network

You can take two devices and start a network using the CLI.

1. Connect to the coordinator of the network using the Ember Desktop console or a simple telnet program. If the device exposes its CLI on port 1, you can connect to it by telnetting to port 4901. Once connected to the device, use the network form command to form the network.

   ```
   Device 1> network form 11 2 0x00aa
   ```

2. Use the network pjoin command to permit joining, so that new devices can come into the network:

   ```
   Device 1> network pjoin 0xff
   ```

3. Connect the second device to the created network using the network join command:

   ```
   Device 2> network join 11 2 0x00aa
   ```

11.2.2 Example 2: Sending an attribute read

Once a network has been formed, you can send messages within the network using the CLI. For example, read the basic cluster's ZCL version using the global read command.

1. Create the command by populating the Ember application framework's messaging buffer.

   ```
   Device 2> zcl global read 0 0
   ```

   This command writes the global read for cluster id 0, attribute id 0 into the messaging buffer.

2. Send the global read command to the device using the send command. The send command takes three arguments: the two-byte node ID to which the message should be sent, the sending endpoint, and the destination endpoint.

   ```
   Device 2> send 0x0000 1 1
   ```

   This command sends the global read command from Device 2 to Device 1, which is the coordinator of the network and thus has the short node id 0x0000.

11.2.3 Example 3: Sending a cluster command

Many of the core clusters to the ZCL have CLI commands built into the Ember application framework. For instance, the identify command allows you to create a ZCL identify command using the ZCL identify CLI command, and send it using the send command.

```
Device 2> zcl identify id 30
Device 2> send 0 1 1
```

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 you enter 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. If you make a mistake in the command, you have the opportunity to re-enter the command before sending it.

In order to send the ZCL command over the air, use a separate send command provided by the CLI.

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

Whenever you send a message, the node through which the message is sent reports which cluster is being transmitted. Likewise, whenever you receive a message, it gives a printout of what it receives.

11.3 Command Line Reference

Note The command line reference that was included in this document has been removed in favor of the reference provided in document 120-3023-000, the *Ember Application Framework API Reference*. The reference included in that document is generated directly from the command line code and as a result is more up-to-date than the reference provided in this document.

12 The Debug Printing Interface

Ember application framework includes a granular debug printing interface. Debug printing, as well as some generic debug printing options like application, core, and custom debug printing, may be controlled on a per cluster basis. Debug printing for each area can be turned on and off in the Ember AppBuilder interface and is controlled by #define values in the application header.

Each debug printing option corresponds to a set of macros used for that specific area of debug printing. For instance, if the “Core” debug printing is turned on, the following macros are populated.

emberAfCorePrint(…) - prints a single line without a carriage return

```
Example: emberAfCorePrint(“node id: %2x”, nodeId);
```

emberAfCorePrintln(…) - prints a single line with a carriage return

```
Example: emberAfCorePrintln(“node id: %2x”, nodeId);
```

emberAfCoreFlush() - flushes the serial buffer

This function should be used if a lot of printing is taking place.

```
Example: emberAfCoreFlush();
```

emberAfCoreDebugExec(x) - includes x in the code

This can be used to wrap code segments like function calls that should only execute when core debug is turned on.

```
Example: emberAfCoreDebugExec(emAfPrintStatus("Success", "Set Failed", ezspStatus));
```

emberAfCorePrintBuffer(buffer, len, withspace) – prints a given buffer as a series of hex values

This is a useful print function for printing out the contents of a given buffer.

```
Example: emberAfCorePrintBuffer(buffer, 0xff, TRUE);
```

emberAfCorePrintString(buffer) – prints a given buffer as a string of characters

This is a useful print function for printing out the contents of a given buffer.

```
Example: emberAfCorePrintString(buffer);
```

13 Multi-Network Support

As of EmberZNet PRO 4.7, the stack supports multiple network configurations (see document AN724, Designing for Multiple Networks on a Single ZigBee Chip, for detailed information on this functionality). The Ember application

framework builds on this feature and provides additional network management functionality to help customers easily build and deploy multi-network devices. The framework automatically switches between the networks when sending and receiving messages, before calling callbacks, and before triggering event handlers for certain types of events. These features reduce the burden of network switching for customers.

Note: For EmberZNet PRO 4.7, multi-network support is limited to two networks. More than two networks will be supported in the future.

13.1 Network Contexts

The stack has separate network contexts for APIs and for stack handlers. The "current network," which applies to all stack API calls, is exclusively managed by the application. The "callback network," which applies to all stack handlers, is exclusively managed by the stack.

The application manages the current network through the use of emberGetCurrentNetwork and emberSetCurrentNetwork. When the application calls any stack API, the stack acts in the context of the current network. For example, calling `emberSetCurrentNetwork(0)` followed by `emberGetNodeId()` returns the node id of network 0. Once set, the current network remains constant until changed by the application through a subsequent call to emberSetCurrentNetwork. The current network is never changed by the stack.

Each invocation of a stack handler applies to a particular network. Before calling any stack handler, the stack sets the callback network to the appropriate network. For example, when a message arrives on network 0, the stack sets the callback network before calling emberIncomingMessageHandler. The application can query the callback network through the use of emberGetCallbackNetwork. In this example, in emberIncomingMessageHandler, `emberGetCallbackNetwork()` returns 0. The callback network only applies within the context of a stack handler. The application cannot change the callback network.

At any given time in the lifecycle of the application, the current network and the callback network may be different. The stack leaves management of the network contexts entirely to the application. To reduce the complexities of multi-network devices, Ember application framework manages the network contexts on behalf of the application.

13.1.1 The Callback Network

In each stack handler it implements, the framework sets the current network to the callback network before performing any of its own processing and before invoking any of its own callbacks. More specifically, at the beginning of each handler, the framework performs an operation equivalent to `emberSetCurrentNetwork(emberGetCallbackNetwork())`. In this way, any code called as a result of a stack handler always acts in the context of the handler itself. For example, if the device receives a GetCurrentPrice message on network 0, the stack calls emberIncomingMessageHandler with the callback network set to 0 and the framework immediately sets the current network to network 0. As a result, when the framework parses the message and ultimately calls emberAfPriceClusterGetCurrentPriceCallback to pass it to the application, the network context is set up to match that of the incoming message. This eliminates the need for the application to check and set the current network itself whenever a message is received.

13.1.2 The Current Network

Because endpoints are central to many of operations performed by a ZigBee device, each endpoint in the application is assigned to a network. Multiple endpoints can be assigned to the same network, but each endpoint belongs to exactly one network. The network assignments are set during configuration in Ember AppBuilder and are not changeable at runtime. When preforming a task on an endpoint, the framework automatically sets the current network to the network of the endpoint.

When sending a message using any of the emberAfSend APIs, the framework, before submitting the message to the stack, automatically switches to the network of the source endpoint. This enables applications to construct and

send messages without the need to check and set the current network. The application cannot override the network on which the message will be sent. Multi-network applications must correctly set the source endpoint for all outgoing messages. Failure to set the proper endpoint results in errors or unexpected behavior.

As messages are received, the framework first verifies that the destination endpoint belongs to the network from which the message arrived. If there is a mismatch, the message is dropped with no ZCL-level response. This ensures proper segregation between the various networks. Otherwise, as mentioned previously, the framework switches to the incoming network so that all subsequent actions are carried out in the context of the appropriate network.

The framework also sets the current network before cluster, endpoint, or network events are triggered. Cluster events are available to the application and can be used to perform operations that are specific to a particular cluster and endpoint. Endpoint and network events are used in plugins and are intended to be used when the plugin does not implement a ZCL cluster or when it provides some auxiliary function to another cluster plugin. For example, if endpoint 1 is assigned to network 0 and the endpoint implements the Price cluster server, the framework sets the current network to 0 before calling emberAfPriceClusterServerTickCallback for endpoint 1. By switching networks before triggering these event handlers, applications and plugins are able to immediately begin their processing without the need to check or set the network.

13.1.3 Switching Networks

The framework maintains active networks in a last in, first out (LIFO) data structure. When it switches networks, the framework actually "pushes" the new network into the LIFO list by calling emberAfPushCallbackNetworkIndex, emberAfPushEndpointNetworkIndex, or emberAfPushNetworkIndex. For example, at the beginning of its implementation of emberIncomingMessageHandler, the framework calls emberAfPushCallbackNetworkIndex to switch the current network to the callback network. Similarly, when sending a message, the framework calls emberAfPushEndpointNetworkIndex with the source endpoint of the message. After completing its processing, the framework "pops" the most recent network from the LIFO list and switches back to the previous network. This is accomplished by calling emberAfPopNetworkIndex. For example, at the end of both emberIncomingMessageHandler and emberAfSendUnicast, the framework calls emberAfPopNetworkIndex to switch to whichever network was set previously.

If the framework has to switch network for any reason, it always restores the previous network when it has finished its processing. In this way, the application is assured that the current network remains constant before and after a call to a framework API. The framework relies on the same guarantee, which means that the application and all plugins must exclusively use the emberAfPush APIs in conjunction with emberAfPopNetworkIndex when switching networks. Using emberSetCurrentNetwork directly will bypass the LIFO list and result in errors. Additionally, every successful push must be matched with a corresponding pop. Failure to follow this paradigm will result in errors.

13.2 Configuration

Ember AppBuilder is used to enable and configure multi-network functionality in framework applications. The steps to configure a multi-network application, which are described in detail below, are as follows:

- Create one or more networks
- Assign endpoints to networks
- Set the default network
- Configure the ZigBee device type for each network
- Configure the security profile for each network

Endpoints are assigned to networks on the "ZCL cluster configuration" tab in Ember AppBuilder. By default, all endpoints are assigned to a network named “Primary.” To create a new network and assign an endpoint to it, click in the “Network” column of the endpoint row and then click on the button that appears next to the network name. A dialog box appears in which a new network can be created by clicking on “Create new network” and providing a

unique name in the accompanying text field. To assign an endpoint to an existing network, select “Use network” in the same dialog box and choose the network name from the list of networks. After clicking “OK,” the list of endpoints will reflect the new network assignment.

For ease of use, Ember AppBuilder generates friendly identifiers for each network based on the user-defined name. These identifiers can be used in place of literals in the code. For example, for a network named "SleepyEndDevice," the application could call emberAfPushEndpointNetworkIndex(EMBER_AF_NETWORK_INDEX_SLEEPY_END_DEVICE) to switch to that network.

An unlimited number of networks can be created using the "ZCL cluster configuration", but endpoints may only be assigned to at most two networks. Having more than two active endpoints in an application is an error and Ember AppBuilder will not generate a configuration for such an application. In additional, as discussed previously, each endpoint belongs to one network and each network can contain multiple endpoints. The endpoint numbers must be unique across the entire device. For example, networks 0 and 1 cannot both contain an endpoint with number identifier 1.

Once a network is created, it can be configured on the "Stack" tab in Ember AppBuilder. The network widget lists each network and shows whether it is active or inactive. Active networks are shown in black text and are those networks to which endpoints have been assigned. Inactive networks are shown in grey italics and are those network to which no endpoints have been assigned. Inactive networks also have “(unused)” appended to their name. Ember AppBuilder saves inactive networks in the application configuration file but it will not generate any code related to such networks. In addition to showing whether a network is active or inactive, the network widget also indicates which network is the default network. The default network is shown in bold and has “(default)” appended to its name. In the framework, the default network is the one used for the initial push. Unless another push occurs, the default network applies to all stack API calls. In addition to an identifier for its friendly name, as previously described, Ember AppBuilder generates EMBER_AF_DEFAULT_NETWORK_INDEX with the index of the default network. To change the default network, select an active network and click on the “Make Default” button.

The ZigBee device type and security profile can be configured for each network. To change the ZigBee device type, click in the “ZigBee Device Type” column of the network row and then click on the button that appears next to the device type. A list of available ZigBee devices will appear. To configure the security profile, click in the “Security Type” column of the network row and then click on the button that appears next to the profile. There are limitations to how the ZigBee device type and security profile can be configured for multiple networks, as described below.

13.2.1 ZigBee Device Types

In multi-network devices, one network must be configured as a sleepy end device. The other networks may be configured as a coordinator, router, end device, or sleepy end device. This restriction is enforced during configuration in Ember AppBuilder and cannot be changed at runtime. The device type for each network is configured on the "Stack" tab in Ember AppBuilder.

The new End Device Support plugin supports polling, sleeping, and rejoining for single-network end devices and all multi-network devices. The plugin will short or long poll as appropriate, sleep when possible, and rejoin whenever it finds itself operating without a network.

For single-network devices, polling is similar to past releases, except that it is in a plugin now instead of being built in to the core framework. The plugin will short poll when it is expecting data and long poll otherwise. For multi-network devices, the plugin will short or long poll on a per-network basis. For example, if data is expected on network 0 but not on network 1, the plugin will short poll on network 0 while long polling on network 1. The plugin allows different polling intervals for each network. For example, network 0 may short poll every second while network 1 short polls every other second. The intervals are configurable at runtime through the emberAfSetShortPollIntervalQsCallback and emberAfSetLongPollIntervalQsCallback APIs. As with all APIs that

affect individual networks, the application should push the appropriate network before calling either API, and afterward pop to restore the previous network.

Sleeping is also similar to past releases for single-network devices. The plugin will stay awake, idle, or sleep as appropriate. For multi-network devices, the plugin will only sleep if all networks are sleepy. For example, a device configured as a router on one network and a sleepy end device on the other will not sleep because the router is expected to remain awake. For devices configured as sleepies on all networks, the plugin will sleep when all networks are able to do so. The most restrictive per-network sleep policy determines the per-device sleep policy.

Finally, rejoining works the same as it has in past releases for single-network devices. The plugin will attempt to rejoin the network after a series of failed data polls. For multi-network devices, the process is similar, with rejoining being performed on a per-network basis as determined by per-network data polls.

Without the End Device Support plugin or equivalent functionality provided by the application directly, the application will not poll, sleep, or rejoin at all. Because of the complexity of managing polling, sleeping, and rejoining, all customers building single-network end devices and all customers building any multi-network device are strongly encouraged to use the plugin.

13.2.2 Security Configuration

While Ember AppBuilder and Ember application framework continue to support the "None" and "Custom" security profiles, only the "Smart Energy" (SE) and "Home Automation" (HA) profiles are supported with multiple networks. Each network can have a different security profile, but only the combinations SE/SE and SE/HA are supported in this release. This restriction is enforced during configuration in Ember AppBuilder and cannot be changed at runtime. The security profile for each network is configured on the "Stack" tab in Ember AppBuilder.

Security is primarily handled by the framework, although the application does have control over the initial security settings used when forming and joining devices. Using the new emberAfSecurityInitCallback, the application can override the initial security bitmask and the extended security bitmask on a per-network basis.

14 Sleepy Devices

14.1 Introduction

The Ember application framework contains support for sleepy end devices. A sleepy end device is a device on the ZigBee network that spends most of its life powered down and only powers up the processor when it needs to do something specific like interpret a GPIO interrupt or poll its parent to see if there are any messages waiting for it on the network.

Note As of stack release 4.7 all of the sleep-related code has been moved to a plugin called the “End Device Support Plugin.” All options mentioned in this chapter are available for configuration within that plugin.

14.2 Polling

Sleepy end devices do not receive data directly from other devices on the network. Instead they must poll their parent for data and receive the data from their parent. The parent acts as a surrogate for the sleepy device, staying awake and buffering messages while the child sleeps. As a result, the sleep/wake cycle of the sleepy end device is governed by two important timeouts on the ZigBee network: the APS retry timeout (7.8 seconds) and the End Device Poll timeout (defined by the parent defaults to 5 minutes). These two timeouts correspond to two polling intervals on the sleepy end device: the SHORT_POLL and the LONG_POLL intervals. These intervals are sometimes referred to as the “nap duration” and “hibernation duration” respectively. So when a device is in a state where it is continually sending polls out on the SHORT_POLL interval it is considered to be “napping,” due to the

fact that it is continually waking up after a very short period to poll. When a device is sending out polls on the LONG_POLL interval it is said to be “hibernating,” due to the fact that it is sleeping for a longer interval.

When a device needs to be responsive to messages being sent to it from the network, it goes into a state where it polls its parent on the SHORT_POLL interval (napping). This ensures that any messages received by its parent will immediately be retrieved by the sleepy end device and processed. When the device no longer needs to be as responsive on the network it returns to a state where it polls its parent on the LONG_POLL interval (hibernating) which ensures that the child will remain alive in its parent’s child table but will not be responsive to the network.

The time during which the sleepy end device is polling at an augmented rate based on the SHORT_POLL interval is referred to as the “Short Poll Mode,” “Fast Poll Mode,” or simply “Napping.” All of these terms mean the same thing. The sleepy device is polling its parent faster than the 7.68 seconds allowed for an end devices parent to hold onto a message for the end device. Generally the SHORT_POLL interval will be something less than 1 second to ensure that all messages sent to the parent are pulled off in an orderly fashion, since the parent is only required to hold onto a single message. If the messages are not retrieved from the parent quickly enough they may be overwritten by other incoming messages for the same child or some other child. For more information on Fast Polling see the section below entitled Forcing “Fast Polling”.

14.2.1 The SHORT_POLL Interval

The short poll interval is the amount of time that an end device may wait before polling its parent when it is in the process of sending or receiving a message. This interval must be shorter than the Indirect Transmission Timeout (standardized at 7.68 seconds for ZigBee PRO networks). This is because the end device must send an APS ACK back to the sending device before the sending device decides to resend the message. The end result is that, in order for sleepy end devices to reliably communicate with other devices on the network, they must know when they are in the process of sending or receiving a message and must wake and poll their parent for data within the short poll interval until the message transaction is complete.

14.2.1.1 Setting at Compile Time

The short poll interval is defined in quarter seconds in the framework by EMBER_AF_SHORT_POLL_INTERVAL. The Ember AppBuilder interface has no GUI widget to allow users to manipulate this value. It defaults to one unit which is equal to one quarter-second. If you wish to change the default EMBER_AF_SHORT_POLL_INTERVAL you may add a definition for it in the macros section of the “includes” tab in the application configuration.

14.2.1.2 Setting at Runtime

Within the application framework, the EMBER_AF_SHORT_POLL_INTERVAL is assigned to a global variable called emberAfNapDuration, which can be modified at runtime using the EMBER_AF_SET_NAP_DURATION (int32u duration) macro.

Note As of stack release 4.7 the variable emberAfNapDuration is deprecated and has been replaced by the callback emberAfGetShortPollIntervalQsCallback(). Likewise, the function EMBER_AF_SET_NAP_DURATION is deprecated and has been replaced by the function emberAfSetShortPollIntervalQsCallback(int16u shortPollIntervalQs).

14.2.2 The LONG_POLL Interval

The long poll interval is the amount of time that an end device may wait before polling its parent when it is otherwise inactive. This interval should, but is not required to, be shorter than the “End Device Poll Timeout,” which is the amount of time an Ember parent device will wait to hear from its child before removing it from its child tables. The default end device poll timeout for Ember devices is 320 seconds or just over 5 minutes.

Note: The ZigBee protocol does not offer a standard way to timeout entries in a child table. In place of this, several heuristic mechanisms exist for aging entries in a child table. For instance, if a parent hears a device that it thinks is its child interacting with another parent or being represented by another parent, it may remove the entry from its child table. Silicon Labs has developed a more deterministic mechanism for child aging called the “End Device Poll Timeout.” An Ember parent expects that children will “check in” with their parents within the end device poll timeout. If they do not, it assumes that they have gone away and removes them from its child tables. The End Device Poll Timeout is defined in stack/include/ember-configuration-defaults.h

The end device does not get to configure the end device poll timeout on its parent and there is no agreed upon protocol for communicating the End Device Poll Timeout value between parent and child. In place of this, Silicon Labs has a configured an assumed end device poll timeout on both parent and child. This value is defined in stack/include/ember-configuration-defaults.h.

Depending on its sleep characteristics, battery life considerations, the child may wish to sleep past the assumed end device poll timeout. It is free to do this. However, if it does, it must repair the network connection with its parent before interacting with the network again. Generally a device that is likely to do this should check the state of the network when it wakes up to see if any repair is necessary before sending data. A sleepy device should never wake and assume that its parent is still there, unless it knows for certain that its parent is configured with a mutually agreed upon End Device Poll Timeout that it is obeying. For more information on the end device poll timeout on Ember devices see the configuration header file located at stack/include/ember-configuration-defaults.h.

14.2.2.1 Setting at Compile Time

The long poll interval is defined in quarter seconds in the framework by EMBER_AF_LONG_POLL_INTERVAL. The long poll interval can be manipulated within the stack configuration tab. EMBER_AF_LONG_POLL_INTERVAL can also be defined in the “Includes” section of the stack tab.

14.2.2.2 Setting at Runtime

Within the application framework, the EMBER_AF_LONG_POLL_INTERVAL is assigned to a global variable called emberAfHibernateDuration, which can be modified at runtime using the EMBER_AF_SET_HIBERNATE_DURATION(int32u duration) macro.

Note As of stack release 4.7 the variable emberAfHibernateDuration is deprecated and has been replaced by the callback emberAfGetLongPollIntervalQsCallback(). Likewise, the function EMBER_AF_SET_HIBERNATE_DURATION is deprecated and has been replaced by the function emberAfSetLongPollIntervalQsCallback(int16u longPollIntervalQs).

14.2.3 What Values Should I Set for the Short and Long Poll Intervals?

The Short Poll Interval should be less than the Indirect Transmission Timeout of the parent to prevent lost data/ACKs (< 7.8 s). The Long Poll Interval should be less than the End Device Poll Timeout of the parent (assuming the parent implements an End Device Poll Timeout) to prevent the parent from aging out the end device due to inactivity. By default the Ember stack ships with a 5 minute End Device Poll Timeout. The manufacturer can change the End Device Poll Timeout as they wish. There is no standard way for routers to report their chosen End Device Poll Timeout to their children and it is not required for routers to implement child aging in the ZigBee specification. As a result, if a device implements a LONG_POLL_INTERVAL that is slower than 5 minutes, we recommend that the device check its network status before spontaneously sending messages through its parent. You want the device to make sure that the connection to the parent is up before it sends a message. If the network

is not up, the device should perform a network rejoin to make sure that it has a parent before sending any message out over the air.

14.2.4 Forcing “Fast Polling”

Fast Polling is the state during which the stack actively polls its parent device faster than the 7.68 second child message timeout interval. The Ember application framework polls at the rate defined by the SHORT_POLL interval when it is in this mode. The Ember application framework automatically keeps the stack in the fast poll mode during the sending and ACKing of an APS message. When a device sends a message that is part of a series of application-level request/responses, as is the case in Smart Energy Registration, it must keep the device in fast poll mode until the entire transaction is completed.

The Ember application framework can ensure that the application stays in short poll mode for as long as the application requires by setting a flag in the emberAfCurrentAppTasks mask. In order to do this, create your own flag for the emberAfCurrentAppTasks that fits in with what is available according to the named masks in app/framework/include/af.h. The top 16 bits of the emberAfCurrentAppTasks mask are reserved for customer use. Once you have chosen a flag for your application, you may use the emberAfAddToCurrentAppTasks and emberAfRemoveFromCurrentAppTasks functions to add and remove your flag. If the flag is present in the emberAfCurrentAppTasks global bitmask, the application does not allow the stack to back into hibernation mode and the stack stays in short poll mode, during which it uses the SHORT_POLL interval to determine how quickly to poll the parent. The usage of this API is also documented in app/framework/include/af.h.

14.2.5 Using “Fast Polling” to Complete a Complex Transaction

Sometimes a sleepy device needs to stay in fast poll mode while sending a complex series of messages that constitute a complete application level transaction with another device. The general strategy for this type of interaction on a sleepy end device is as follows:

1. Sleepy end device A needs to perform a series of messages with device B, which we will simply call a transaction.
2. Sleepy end device A creates an event that will serve as a timeout for the application level transaction. We will call this the transaction timeout event.
3. Sleepy end device A starts the event and sends the first message to device B.
4. If the message is an APS message, sleepy end device A will automatically stay in short poll mode until the APS Ack comes back from the responding device.
5. If the message is a ZCL command, sleepy end device A will also automatically stay in short poll mode long enough to give device B a chance to send any application level command response required by the ZCL
6. Sleepy end device A continues with its series of messages back and forth to device B until the whole transaction is complete.
7. When the final message of the transaction is completed with device B, Sleepy end device A removes the flag from emberAfCurrentAppTasks thereby allowing the device to naturally go back to using the hibernate or LONG_POLL period for sleeping.
8. If device A and B are not able to complete their transaction as expected, Sleepy End Device A removes the flag from emberAfCurrentAppTasks when the transaction timeout event set up in step #1 fires.

14.2.6 Difference in Polling on SOC and Host+NCP Models

The requirements of polling result in different sleep patterns for the System-On-Chip (SOC) and the Host + Network Co-Processor (NCP) models. In the Host + NCP model, it is the NCP that is responsible for polling at the SHORT_POLL and LONG_POLL intervals. The only responsibility of the host processor is to tell the NCP how frequently to poll. Other than that the host may sleep indefinitely or until there is some internal event, a GPIO interrupt or the NCP receives a message that it passes to the host for processing. Conversely, the SOC itself is responsible for polling its parent, so it must be sure to wake within the SHORT_POLL and LONG_POLL intervals in

order to do so. The Ember application framework uses the internal event mechanism on the SoC to schedule polling. On the host, it sends a message down to the NCP to tell it when to poll.

14.3 Sleeping and the Event Mechanism

The Ember application framework automatically checks with the event mechanism to see when the next application event is scheduled. The Ember application framework never sleeps through an event. The sleep period is always shorter than the amount of time to the next application event within the framework. On the SoC the amount of time that a device will sleep is generally governed by the SHORT_POLL and LONG_POLL intervals, since the polling event is also an event within the Ember application framework. On the host, the processor will attempt to sleep until the next application event.

14.3.1 Never Use Ticks On a Sleepy End Device

All application events should be scheduled through the event mechanism using either custom or cluster events on a sleepy end device. This is because the event mechanism provides a central repository for the sleep handling code so that it knows how long it can sleep. If you rely on the emberAfMainTickCallback to fire frequently enough to handle application events on your sleepy, you will be forced to wake the sleepy on an artificially short interval so that the emberAfMainTickCallback can be serviced. For an example of how to write a sleepy application using custom events see the sleepy gas meter sample application located at: app/framework/sample-apps/se-meter-gas-sleepy.

14.4 End Device Parent Rediscovery

If an end device loses contact with its parent it will automatically begin to rejoin the network either with the existing parent or a new parent by calling emberAfStartMove. The emberAfStartMove function schedules a “move” event in the Ember application framework’s event scheduling mechanism with the following characteristics:

- When the move event fires, the device calls emberFindAndRejoinNetwork.
- The move event is automatically rescheduled so that a network rejoin will be attempted every 10 seconds until EMBER_AF_REJOIN_ATTEMPTS_MAX is reached.
- If EMBER_AF_REJOIN_ATTEMPTS_MAX is set to 0xff (default) the rejoin will be attempted every 10 seconds until a network is found.
- The first attempt to rejoin the network is always performed with security on. Each subsequent attempt is performed with security off.

This orphan behavior can obviously have an impact on the life of a battery-powered device. If you would like to limit the number of rejoin attempts a device performs before it gives up you can set EMBER_AF_REJOIN_ATTEMPTS_MAX to something other than 0xff by adding an entry in the Additional Macros section of the Includes tab within your Ember AppBuilder configuration.

14.5 Sleepys and the CLI

It is very difficult to interact with a sleepy end device on the Command Line Interface (CLI) when it is sleeping. For this reason we used to keep sleepy end devices awake until they were connected to a network. We do not do this any longer; however, if you would like your sleepy end device to stay awake when it is not connected to a network you can do so by including the –D define EMBER_AF_STAY_AWAKE_WHEN_NOT_JOINED in the custom macros section of the includes tab within Ember AppBuilder.

The other alternative is to provide a button handler that toggles the device between a default wake and sleep state. For an example on how this is done see the SeMeterGasSleepy sample application located at app/framework/sample-apps/se-meter-gas-sleepy/

14.6 Processor Idling and the Application Framework

The Ember application framework implements processor idling for sleepy devices as of EmberZNet PRO stack release 4.6. This feature augments power savings for sleepy devices by idling the processor during times that there are no events happening. This means that a sleepy device will not continually run through the application's main loop when it is awake. Instead the processor will idle until it receives an interrupt either from an external line or a scheduled event. Once each event has been handled, it is marked as ready to idle. The processor will then wait until the next internal or external interrupt before running through the application's main loop again.

Some typical examples of when a sleepy device can save energy by idling include:

- While a packet, such as a data poll, is being transmitted from a sleepy device, the CPU is usually just waiting for the transmission to finish and can idle.
- While waiting for the crystal to stabilize, the CPU eventually runs out of initialization and calibration operations to do, so it can idle.
- While waiting for the ACK to a transmitted packet. The radio still needs to be on in receive mode, so the processor can't deep sleep, but can safely idle.

15 Application Framework Plugins

15.1 Introduction

The Ember application framework contains support for plugins. A plugin is an implementation of a piece of functionality within the framework through the framework's callback interface. The Ember application framework ships with default implementations of many of the clusters, such as key establishment and price among others. This chapter documents some of the plugins shipped with the Ember application framework.

15.2 Creating Your Own Plugins

Plugins are encapsulated implementations of the callback interface. If a specific plugin does not satisfy your needs you have two options.

1. Implement the associated callbacks directly in your application
    1. Disable the plugin and implement the callbacks you need for your application.
    2. This will add the callbacks into your application_callbacks.c file.
2. Create your own plugin from the original
    1. Go to the plugin directory within the stack install located at app/framework/plugin.
    2. Copy the plugin contents into a new directory inside app/framework/plugin.
    3. Each plugin includes a configuration file called plugin.properties. This file is used by Ember AppBuilder to display the plugin within the configuration pane and manage dependencies within the framework. At the very least, you must change the name of the plugin in the plugin.properties file so that you can recognize it in Ember AppBuilder.
    4. Once you do this, you must restart Ember AppBuilder so that your new plugin is picked up when Ember AppBuilder scans the stack install directory.

15.3 Over the Air Upgrade (OTA) Plugins

The Over-the-air Bootload cluster is a large piece of functionality in the Smart Energy 1.1 specification. It involves a number of modules in order to support software implementations on all Ember platforms and for both the client and server.

This section details each of the different pieces and describes their function in the Ember application framework.

15.3.1 Architecture

This section explains the architecture of the cluster and where developer code fits into the Ember application framework.

The ZigBee Over-the-air Bootload (OTA) cluster provides a common way for all devices to have a manufacturer-independent method to upgrade devices in the field. The ZigBee OTA cluster only supports application bootloaders where a device has the capability to download and store the entire image in external storage while still running in the ZigBee network.

The ZigBee OTA cluster defines the protocol by which client devices query for new upgrade images and download the data, and how the server devices manage the downloads and determine when devices shall upgrade after downloading images.

Silicon Labs provides all the cluster code for both client and server to correctly process and respond to all ZigBee OTA messages. In addition, it provides code for managing the stored image(s) and bootloading the target chip.

A number of decisions have to be made about the architecture of the upgrade and how it will be handled. Below are several key questions to answer.

1. What external storage device will be used for the OTA upgrade image?
    1. Silicon Labs provides a few EEPROM driver implementations as well as a POSIX file system (for UART host only).
    2. If a different driver or method is desired, then this code must be provided.
2. Does a client device require multiple upgrade files in order to bootload?
    1. If so, the multiple upgrade files can be co-located within the same ZigBee OTA file transferred over-the-air. However, this requires a storage device that can hold all the upgrade files at the same time.
    2. The ZigBee OTA cluster also supports requesting multiple files, but the client must manage this.
3. How will upgrade files be labeled?

    Each OTA file has a manufacturer ID, an image type ID, and a version number. The value for the manufacturer ID is assigned by ZigBee, but the manufacturer controls the other two values, which can be set to whatever values they want. The choice of what values to use depends on the versioning scheme used by the developer, and how products from the same manufacturer are differentiated.
4. Will image signing and verification be used by client devices?
    1. Although the choice to support the ZigBee OTA cluster is optional for Smart Energy devices, if devices do support the cluster, then manufacturers must digitally sign upgrade images, and their devices must verify the authenticity and integrity of those images.
    2. Manufacturers that use image signing must obtain signing certificates, and embed the EUI64s of allowed signers within the software so downloaded images can be validated.
5. How will bootloading be handled by clients?

    Bootloading is device specific, though Silicon Labs provides sample code to bootload both its SOC and NCP chips. But it is likely the developer will have to provide additional specific code to support their own device.
6. How will the server receive the images to be given out to clients?

    The Ember implementation provides a POSIX server that can serve up OTA files that reside on a file system. If the server is an EEPROM-based system, then some other mechanism must be created to transfer the images to the server, so that those can be served up to ZigBee OTA clients.

15.3.1.1 Generating ZigBee OTA Images

Silicon Labs provides a tool called image-builder that can generate correctly formatted ZigBee OTA images. This tool takes in bootloader files (such as an EBL file) and generates the correct format according to the command-line

input, as illustrated in Figure 5. The tool can also sign the images using the ECDSA signature algorithm as dictated by the ZigBee OTA cluster specification.

For more information on using this tool please consult document AN716, Instructions for Using Image Builder.

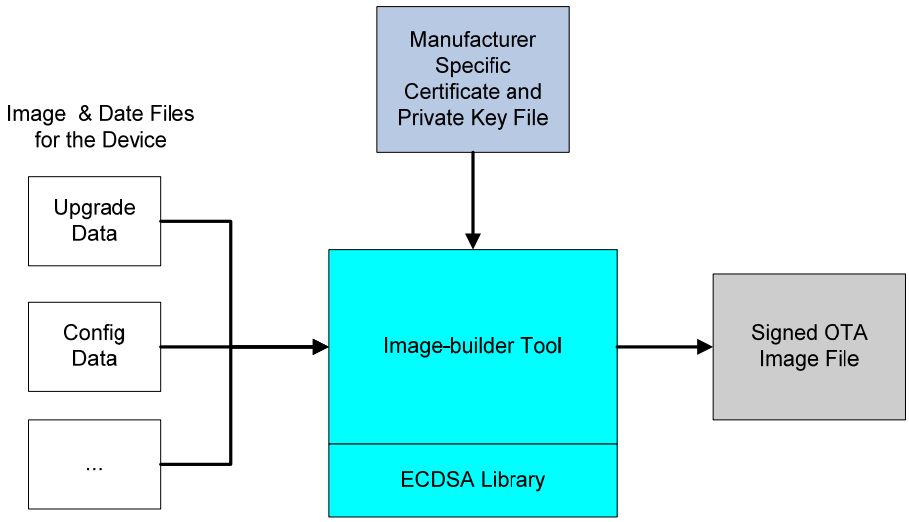

Figure 5. OTA Image Generation

15.3.1.2 Image Signing

The ZigBee Smart Energy Profile requires that OTA files be signed by the manufacturer. Downloaded files must be validated by the OTA client prior to installation. When images are signed the signer's certificate is included automatically as a tag in the file, and a signature tag is added as the last tag in the file.

The EUI of the authorized signing certificates must be embedded in the client's current software image so it can validate that only the certificates pertaining to the manufacturer of the device can sign update images.

For development and or test deployments that want to use signing and OTA images a test certificate can be used from the Certicom Test CA to sign images. The image-builder tool has a test certificate embedded in it for this purpose and by default Ember AppBuilder includes the EUI of that test certificate as an authorized signer.

Note: For generation of production images to be shipped to deployed devices, it is highly recommended that manufacturers use their own certificates issued from the Certicom Production CA to sign images, and specify only these EUIs as authorized signers.

The certificates and private keys used for signing are the same type as certificates and private keys used by Smart Energy devices. However, their use and storage should be handled differently. The following are the differences:

1. Certificates and private keys used for signing should only be used for signing. They should never be put on devices as device-specific certificates and keys. This holds true regardless of whether the device is a test device or a production device.
2. The EUI used for the signing certificate should NOT be used for by any other device or for any other purpose. That EUI should NOT be part of a general pool of EUIs used for production devices.
3. It is recommended that at least three signing certificates with private keys be generated with three different EUIs. Multiple signing certificates allows for deprecating an expired or compromised private key.
4. Devices should be set up to accept all of those EUIs as authorized signers of images. If a single key or certificate is compromised it can be deprecated through a software update, and devices will not accept images signed by that entity. In that case, a new signing certificate should be created to replace the

compromised one and subsequent software releases should set it up to be an authorized signer. In the interim one of the other two alternative signing certificates can be used to sign software updates.

5. Signer private keys should be stored in a secure location with limited access.

Lastly, it should be noted that mixing production device certificates with a test certificate signer (and vice versa) does not work. In other words, if a device has a production certificate from the Certicom Production CA then it can only validate images signed with a production certificate. Similarly devices with test certificates can only accept signers that have certificates issued from the Certicom Test CA.

15.3.2 Plugin Architecture

A diagram of the architecture of the OTA plugins is shown in Figure 6.

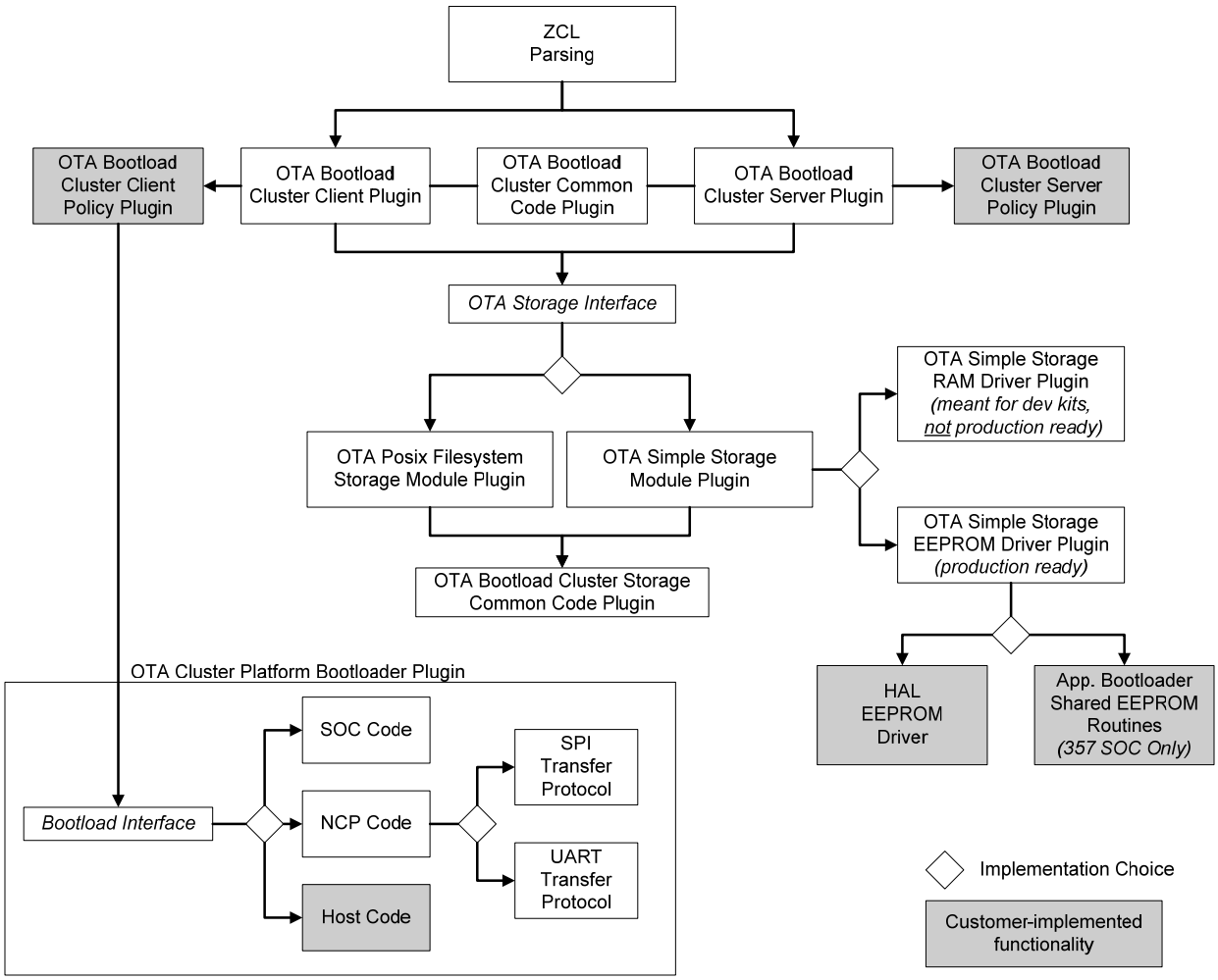

Figure 6. OTA Plugin Architecture

15.3.3 ZCL Parsing

This code is provided by the core Ember application framework and performs the basic parsing and verification of incoming and outgoing messages using the ZigBee Cluster Library.

15.3.4 OTA Bootload Cluster Common Code Plugin

This plugin provides common code to the OTA client and server cluster plugins. It must be enabled if either the **OTA** Bootload Cluster Server Plugin or the OTA Bootload Cluster Client Plugin is enabled.

This plugin has no configurable options.

15.3.5 OTA Bootload Cluster Server Plugin

The OTA Server cluster performs the message parsing of Over-the-air Bootload cluster client commands sent to the server, and generates server commands sent to the clients. It does not handle storage of the OTA files but instead relies upon an external module for that support. Its role is simply to validate the incoming messages and generate the correct replies based on its own supported functionality and the OTA specification.

Silicon Labs provides an Ember application framework plugin that implements the server cluster.

Options	Description
Page Request Support	The OTA Server Cluster plugin can support the optional Page Request feature of the ZigBee OTA cluster. If this option is enabled, the server will answer page requests and send multiple blocks of the download image back to the client.

15.3.6 OTA Bootload Cluster Server Policy Plugin

This module defines how the OTA server reacts when it receives certain requests from the client. The server cluster code calls into this module to ask how certain operations should be handled. For example, when a client is finished downloading a file it sends an *upgrade end request* to the server to ask when it can upgrade to the new image. The server cluster code parses the message and then calls into the server policy code to determine the answer.

Other examples of policies handled by this module include how to respond when a query for the *next* OTA image to download is received, and how to respond when receiving an image block request.

This plugin has no configurable options.

15.3.7 OTA Bootload Cluster Client Plugin

The OTA client cluster performs the message parsing of Over-the-air Bootload cluster server commands sent to the client, and generation of client commands sent to the server. It does not handle storage of the OTA files, but instead relies upon an external module for that support. Its role is simply to validate the incoming messages and generate the correct replies based on its own supported functionality.

Silicon Labs provides an Ember application framework plugin that implements the client cluster. The plugin has optional support for the signature verification feature. When enabled, this checks the ECDSA signature on received OTA files before generating the upgrade end message sent back to the server.

Options	Description
Query OTA Server Delay (minutes)	How often the client queries the OTA server for a new upgrade image.
Download Delay (ms)	How often a new block of data (or page) is requested during a download by the client. A value of 0 means the client requests the blocks (or pages) as fast as the server responds.
Download Error Threshold	How many sequential errors cause a download to be aborted.
Upgrade Wait Threshold	How many sequential, missed responses to an upgrade end request cause a download to be applied anyway.
Server Discovery Delay (minutes)	How often a client looks for an OTA server in the network when it did not successfully discover one. Once a client discovers the server, it remembers that server until it reboots.
Run Upgrade Delay Request (minutes)	How often the client will ask the server to apply a previously downloaded upgrade when the server has previously told the client to wait.
Use Page Request	Selecting this option causes the client device to use an OTA Page Request command to ask for a large block of data all at once, rather than use individual image block requests for each block. If the server does not support this optional feature, then the client falls back to using individual block requests.
Page Request Size	The size of the page to request from the server.

Options	Description
Page Request Timeout (seconds)	The length of time to wait for all blocks from a page request to come in. After this time has expired, missed packets are requested individually with image block requests.
Signature Verification Support	This requires all received images to be signed with an ECDSA signature and verifies the signature once the download has completed. If an image fails verification it is discarded. This verification occurs prior to any custom verification that might verify the contents.
Verification Delay (ms)	This controls how often an ongoing verification process executes. When signature verification is enabled this controls how often digest calculation is executed. Digest calculation can take quite a long time for an OTA image. Other processing for the system may be deemed more important, and therefore we add delays between calculations. This also controls how often custom verification written by the application developer is executed. A value of 0 means the calculations run to completion.
Image Signer EUI64 0	The big endian EUI64 address of a device authorized to sign OTA images for this client. A value of all 0s is ignored.
Image Signer EUI64 1	The big endian EUI64 address of a device authorized to sign OTA images for this client. A value of all 0s is ignored.
Image Signer EUI64 2	The big endian EUI64 address of a device authorized to sign OTA images for this client. A value of all 0s is ignored.

Note: The default value for the Image Signer EUI64 0 option is the EUI64 of the test certificate embedded within the image-builder tool provided by Silicon Labs. Using this default will allow customers to test image signing and verification prior to obtaining production signing certificates from Certicom.

15.3.8 OTA Bootload Cluster Client Policy Plugin

This module controls the OTA cluster client's behavior. It dictates what image information it uses in the query, what custom verification of the image is done by the device, and what happens when the client receives a command to upgrade to the new image.

Silicon Labs provides a plugin that provides a simple implementation of the OTA client policy.

Note: The Manufacturer ID is not set in this plugin, but in the ZCL Cluster Configuration tab of Ember AppBuilder.

Options	Description
Image Type ID	The device's OTA image identifier used for querying the OTA server about the next image to use for an upgrade.
Firmware Version	The device's current firmware version, used when querying the OTA server about the next image to use for an upgrade.
Hardware Version	Devices may have a hardware version that limits what images they can use. OTA images may be configured with minimum and maximum hardware versions on which they are supported. If the device is not restricted by hardware version then this value should be 0xFFFF
Perform EBL Verification (SOC only)	This uses the application bootloader routines to verify the EBL image after signature verification passes.

15.3.9 OTA Storage Plugins

The over-the-air cluster requires a storage device for files received by the OTA clients, or served up by the OTA server. This storage varies based on the device's hardware and design. Therefore this functionality is separated from the core cluster code and accessed through a set of APIs. The interface supports managing multiple files, retrieving arbitrary blocks of data from the files, and performing basic validation on the file format.

Silicon Labs currently provides two main plugins that implement the OTA storage module, the **OTA Storage POSIX Filesystem Plugin** and the **OTA Simple Storage Plugin**.

15.3.9.1 OTA Storage POSIX File System Plugin

This implementation uses a POSIX file system as the storage module to store and retrieve data for OTA files. It can handle any number of files. This plugin is used with an EZSP-based Ember platform (EM260 or EM35x) where the host is connected to the NCP through UART.

This plugin has no configurable options.

15.3.9.2 OTA Simple Storage Plugin

This implementation provides a simple storage module that stores only one file. It uses an OTA storage driver to perform the actual storage of the data in a hardware or software device accessible by the OTA cluster code. When enabled the developer must also select either the **OTA Simple Storage RAM Driver Plugin** or the **OTA Simple Storage EEPROM Driver Plugin**.

The plugin can be used by either an EZSP- or an SOC-based Ember platform.

This plugin has no configurable options.

15.3.9.3 OTA Simple Storage RAM Driver Plugin

This driver provides a RAM storage device for storing files. It is intended only as a test implementation for development on Ember Development kits; it is not intended as production ready code. Prior to integrating external storage hardware into a device, this driver can be useful for examining the basic behavior of the OTA cluster. The storage device has a pre-built OTA image already in place that can be used for downloading but does not actually perform an upgrade.

This plugin has no configurable options.

15.3.9.4 OTA Simple Storage EEPROM Driver Plugin

This driver uses the HAL routines to read and write data from an EEPROM.

For the SOC platforms this module handles the details of re-mapping the image so that it can be read by the bootloader. Existing bootloaders require that the EBL header be the first bytes at the top the storage device, so the code must relocate the OTA header to another location while at the same time providing an interface to the storage code that accesses the OTA file in a linear manner.

Figure 7 illustrates a change in the OTA image on disk.

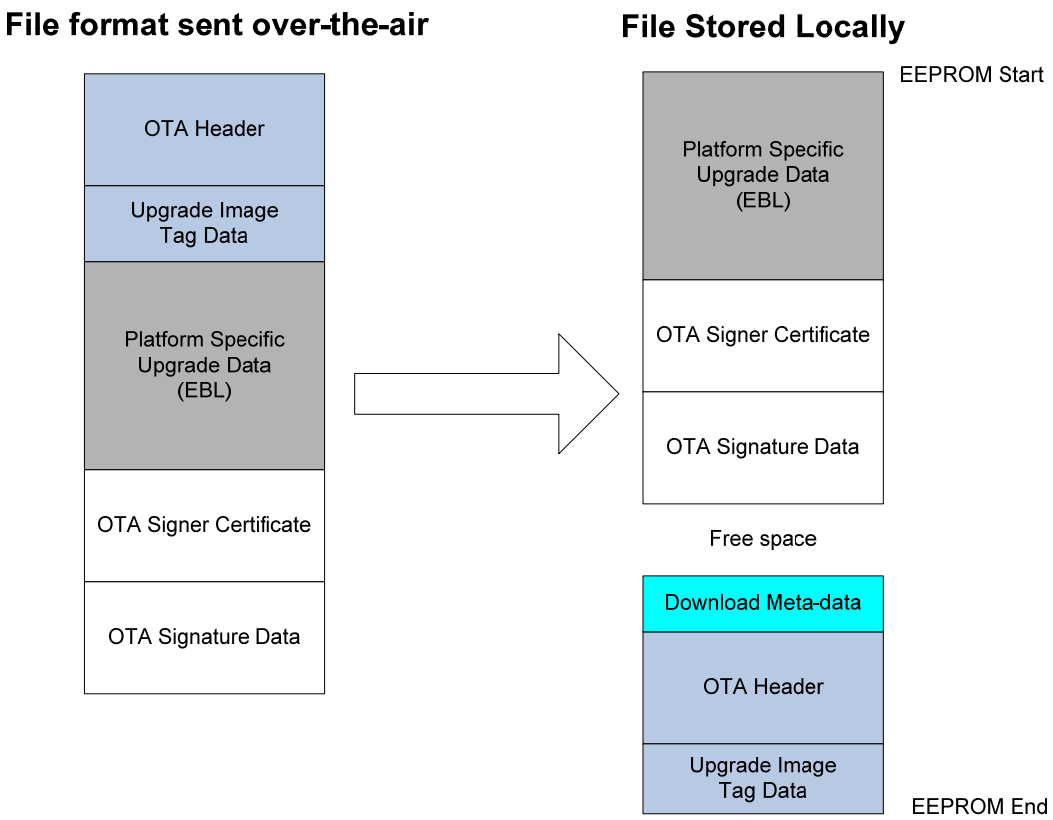

Figure 7. OTA Image Change

Options	Description
SOC Bootloading Support	This option enables bootloading support for SOC devices. When enabled, it will re-map the OTA image file so that the EBL data is at the top of the EEPROM and therefore can be accessed by all existing Ember bootloaders. It requires that the EBL portion of the image is the first TAG in the file. The OTA storage starting offset should be 0 when this is enabled.
EM35x SOC Only: Enable 4.2 Application Bootloader Compatibility Mode	Applications for 35x SOC chips that are running an EmberZNet PRO 4.2 application bootloader must enable this compatibility mode and include a copy of a newer EEPROM driver (EmberZNet PRO 4.3 or later version). Normally the 35x has the ability to use EEPROM driver code shared by both the application and bootloader. However, that EEPROM driver must have support for arbitrary page writes. The Ember drivers included with the EmberZNet PRO 4.2 EM35x application bootloaders did NOT have this support. Therefore this plugin cannot use them and a copy of the driver must be added in the application.
Frequency for Saving Download Offset to EEPROM (bytes)	How often the current download offset is stored to EEPROM, in bytes. If set to 0 it will always be written to EEPROM.
OTA Storage Start Offset	The starting offset for the OTA image storage location in the EEPROM.
OTA Storage End Offset	The last offset for the OTA image storage location in the EEPROM.

Options	Description
EEPROM Device Read-modify-write Support	This checkbox indicates whether the underlying EEPROM storage driver has support for read-modify-write, where a portion of a page of flash can be rewritten without erasing the entire page. If the driver requires a page erase before writing any data this box should not be checked. Before EmberZNet 4.6.2 read-modify-write support was required by the underlying flash driver. EmberZNet 4.6.2 introduced the ability to use parts where a page-erase is required. When designing software for the Ember development boards this checkbox should remain checked since the EEPROM parts require read-modify-write support. For other supported EEPROM parts used in customer designs, refer to Table 5, “Supported Serial Dataflash/EEPROM Remote Memory Parts,” in document UG103.6, *Ember Application Development Fundamentals: Bootloading.*

15.3.9.5 OTA Bootload Cluster Storage Common Code Plugin

This plugin provides code common to all the OTA storage plugins and must be enabled when one of those plugins is enabled.

This plugin has no configurable options.

15.3.10 OTA Cluster Platform Bootloader Plugin

When the client has a completed file downloaded and ready to upgrade, it waits for a command from the server to apply the upgrade. Upon receipt of the command to upgrade, the OTA client cluster code calls into the OTA client policy code to perform the next steps to apply the upgrade.

Silicon Labs provides a single plugin to handle bootloading. The behavior differs depending on the platform on which it is being used. The Ember OTA Client Policy plugin calls into this plugin to perform the actual bootloading.

For the SOC, the bootload code simply calls the HAL routine to execute the application bootloader. The application bootloader then reads from the data stored in external EEPROM, copies that data into the chip’s internal flash, and then reboots.

For the NCP, Silicon Labs provides an implementation that bootloads the NCP through serial UART or SPI bus. This implementation works only with the Ember NCP bootloader provided as part of the EZSP NCP firmware delivery. The implementation executes the bootloader on the NCP, transfers the file from the storage device on the host to the NCP by xmodem, then reboots the NCP.

For the host, developers are expected to write their own code for bootloading a host system connected to an NCP.

15.3.11 OTA Cluster Command Line Interface

15.3.11.1 Client Commands

Table 1. OTA Cluster Client Commands

Command	Description
zcl ota client printImages	Prints all images that are stored in the OTA storage module along with their index.
zcl ota client info	Prints the Manufacturer ID, Image Type ID, and Version information that are used when a query next image is sent to the server by the client.
zcl ota client status	Prints information on the current state of the OTA client download.
zcl ota client verify <index>	Performs signature verification on the image at the specified index.
zcl ota client bootload <index>	Bootloads the image at the specified index by calling the OTA bootload callback.
zcl ota client delete <index>	Deletes the image at the specified index from the OTA storage device.

Command	Description
zcl ota client start	Starts the OTA client state machine. The state machine discovers the OTA server, queries for new images, download the images, and waits for the server command to upgrade.
zcl ota client stop	Stops the OTA client state machine.
zcl ota client page-request <boolean>	Dynamically enables or disables the use of page request if the client turned on support in Ember AppBuilder. By default, if the client enabled page request support in Ember AppBuilder then the client uses the page request command when downloading a file.
zcl ota client block-test	Test harness command. Sends an invalid block request to the client's previously discovered OTA server to verify that the server sends back the correct command.

15.3.11.2 Server Commands

Table 15-2. OTA Cluster Server Commands

Command	Description
zcl ota server printImages	Prints all images that are stored in the OTA storage module along with their index.
zcl ota server policy print	Prints the policies used by the OTA Server Policy plugin.
zcl ota server delete <index>	Deletes the image at the specified index from the OTA storage device.
zcl ota server policy query <int>	Sets the policy used by the OTA Server Policy plugin when it receives a query request from the client. The policy values are: 0: Upgrade if server has newer (default) 1: Downgrade if server has older 2: Reinstall if server has same 3: No next version (no *next* image is available for download)
zcl ota server policy blockRequest <int>	Sets the policy used by the OTA Server Policy plugin when it receives an image block request. The policy values are: 0: Send block (default) 1: Delay download once for 2 minutes 2: Always abort download after first block
zcl ota server policy upgrade <int>	Sets the policy used by the OTA Server Policy plugin when it receives an upgrade end request. The policy values are: 0: Upgrade Now (default) 1: Upgrade in 2 minutes 2: Ask me later to upgrade
zcl ota server notify <dest> <payload type> <jitter> <manuf-id> <device-id> <version>	This command sends an OTA Image Notify message to the specified destination indicating a new version of an image is available for download. The payload type field values are: 0: Include only jitter field 1: Include jitter and manuf-id 2: Include jitter, manuf-id, and device-id 3: Include jitter, manuf-id, device-id, and version All fields in the CLI command must be specified. However if the payload type is less than 3, those values will be ignored and not included in the message.
zcl ota server page-req-miss <modulus>	Test harness command. Sets a module's value that tells the OTA server to artificially skip certain image block responses sent in response to an image page request. This simulates missed blocks that the client will have to request later after the page request has completed. If the number of the block sent by the server is a multiple of the modulus value then it will be skipped.

15.3.12 OTA Client State Machine

Error! Reference source not found. illustrates how the OTA Bootload Cluster client plugin code will behave from start to finish.

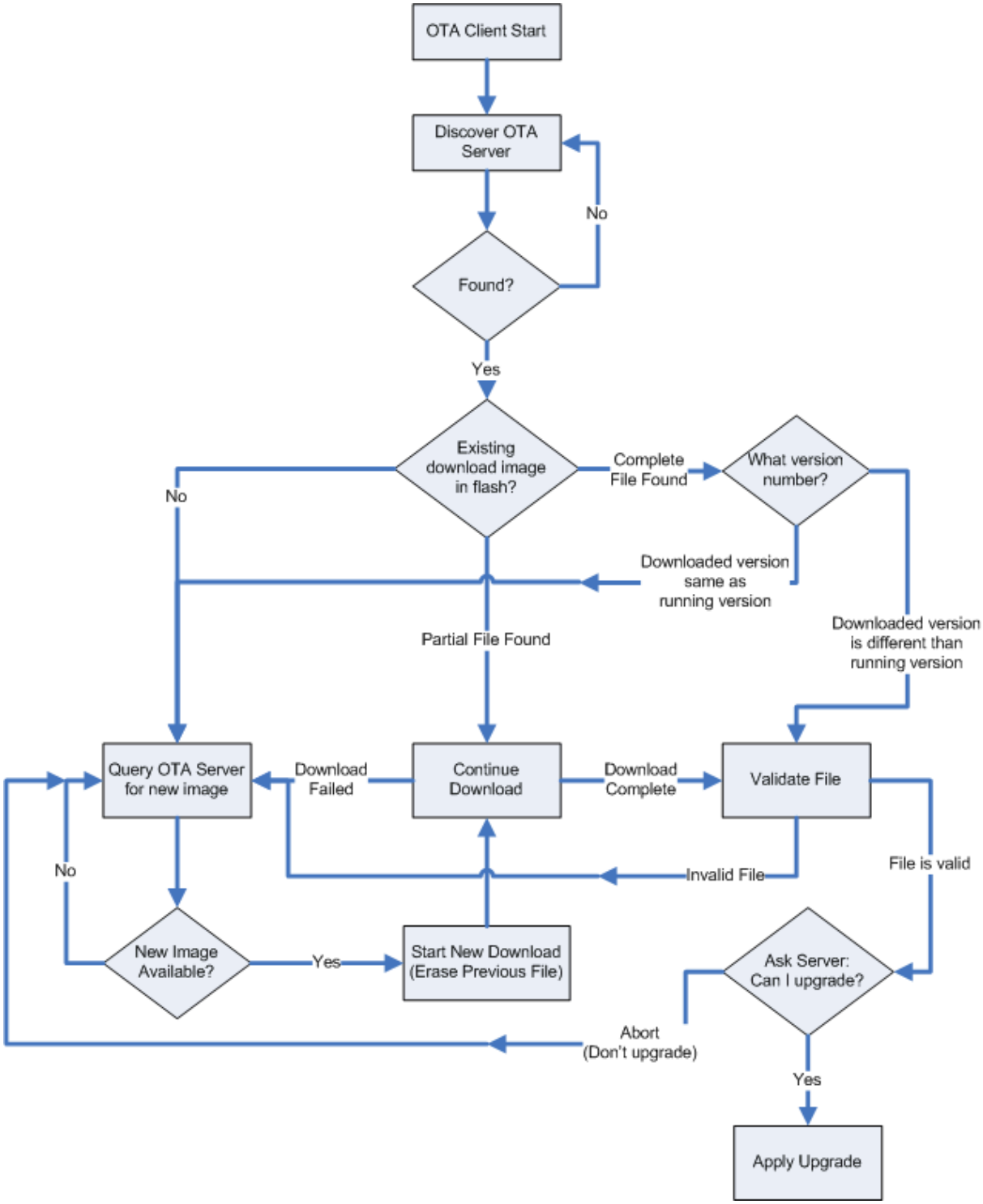

Figure 8. OTA Bootload Cluster Client Plugin Behavior

15.3.13 Example Client and Server Setup

A separate application note (document AN728, ZigBee Over-the-Air Bootload Cluster Client and Server Setup Using EM35x Development Kits) describes a complete client and server setup using these plugins and the Ember 35x Development Kits

15.4 Tunneling Plugin

Tunnels are established between a client and server with a specific protocol (which may be manufacturer-specific) and optional flow control support. The client opens the tunnel by sending a Request Tunnel command to the server. If the server does not support the protocol or flow control, it rejects the tunnel. The tunneling plugins do not deal with protocols at all. Instead, they rely on the application for parsing the raw data and, in the case of the server, for indicating whether particular protocols are supported. The tunneling plugins do not currently support flow control and automatically reject tunnels that request flow control.

Once the tunnel is established, either side can send data with the Transfer Data command. If the sender does not have access to the tunnel, a Transfer Data Error command is sent. If the plugins receive a Transfer Data Error command for a tunnel of which they are a part, they assume that something went wrong and that the tunnel is now closed.

Tunnels can be closed by either the client or server. When the client wants to close the tunnel, it sends a Close Tunnel command to the server. The server can close a tunnel if it has been inactive for some period of time, specified by the Close Tunnel Timeout attribute. If the server closes an inactive tunnel, it does not notify the client, as it is assumed that the client is sleepy in this case. The Close Tunnel Timeout attribute is adjustable by the user through a plugin option.

15.4.1 Tunneling Setup

1. In Ember AppBuilder, select File > New > ZigBee Application Configuration.
2. On the ZCL cluster configuration tab, change the ZCL device type to any of the SE devices.
3. Enable the Tunneling cluster client and/or server.
4. Enable the Generic Tunnel cluster server. NOTE: the spec says that the Tunneling client must include the Generic Tunnel client, but this is an error in the spec.
5. On the Stack configuration tab, in the Debug printing section, enable printing for Tunneling by checking the "Compiled in" and "Enabled at startup" checkboxes next to "Tunneling."
6. Enable printing for Debug by checking the "Compiled in" and "Enabled at startup" checkboxes next to "Debug."
7. On the Stack configuration tab, in the Other settings section, select "Use fragmentation."
8. On the Plugins tab, verify that the Tunneling client and/or server plugins are enabled and adjust any plugin-specific options.
9. Verify that the Generic Tunnel client plugins is enabled.
10. Enable the General response commands plugin.
11. Click Generate to generate the application.
12. In the generated Application_callbacks.c file, modify the emberAfPluginTunnelingServerIsProtocolSupportedCallback stub so that it returns TRUE for any protocols supported by the application.

15.4.2 Tunneling Command Line Interface (CLI)

The framework provides CLI commands for opening and closing tunnels as well as sending data.

15.4.3 Tunneling Client CLI Commands

*** zcl tunneling request <protocol id:1> <manufacturer code:2> <flow control:1>**

* Protocols 0 through 5 are defined by the spec.

* Protocols 6 through 199 are reserved for future use.

* Protocols 200 through 254 are for manufacturer-specific protocols.

* Protocol 255 is reserved.

* The manufacturer code is only used when the protocol is 200 through 254.

* The manufacturer code should be set to 0xFFFF when not used.

* Flow control should be 0 when not used and 1 when used.

*** zcl tunneling close <tunnel id:2>**

* The tunnel id is a unique 16-bit identifier assigned by the server and sent in the Tunnel Request Response command.

*** zcl tunneling transfer-to-server <tunnel id:2> <data>**

* The tunnel id is a unique 16-bit identifier assigned by the server and sent in the Tunnel Request Response command.

* The data is raw protocol data and is not preceded by a length byte like other string types.

*** zcl tunneling random-to-server <tunnel id:2> <length:2>**

* The tunnel id is a unique 16-bit identifier assigned by the server and sent in the Tunnel Request Response command.

* The length is the number of bytes of random data to send.

* The framework and fragmentation library only support messages up to 255 bytes. With the three-byte ZCL header and the two-byte protocol id, only 250 bytes are left for the data.

15.4.4 Tunneling Server CLI Commands

*** zcl tunneling transfer-to-client <tunnel id:2> <data>**

* The tunnel id is a unique 16-bit identifier assigned by the server and sent in the Tunnel Request Response command.

* The data is raw protocol data and is not preceded by a length byte like other string types.

*** zcl tunneling random-to-client <tunnel id:2> <length:2>**

* The tunnel id is a unique 16-bit identifier assigned by the server and sent in the Tunnel Request Response command.

* The length is the number of bytes of random data to send.

* The framework and fragmentation library only support messages up to 255 bytes. With the three-byte ZCL header and the two-byte protocol id, only 250 bytes are left for the data.

15.4.5 Tunneling Current Limitations

Both ends of a tunnel should ensure that only the partner to which the tunnel has been built up is granted read/write access to it. The spec mentions enforcing by checking the MAC address of the sending node. The plugins currently only check node and endpoint IDs and do not check the MAC.

Due to limitations of the framework and the fragmentation library, messages longer than 255 bytes cant be sent or received. The Tunneling cluster client and server are supposed to negotiate a maximum transfer size by reading the Maximum Incoming Transfer Size and Maximum Outgoing Transfer Size attributes from the Generic Tunnel cluster. The Generic Tunnel plugin sets these attributes, but Tunneling plugins do not read them from the remote node.

The spec says that the Protocol Address attribute in the Generic Tunnel server cluster should be set to the complex protocol address of the Smart Energy device represented on that endpoint. The plugins do not set this attribute, although it can be written manually from the CLI or through application code.

Ember Desktop does not currently include decoders for the Tunneling cluster.

16 Extending the ZigBee Cluster Library (ZCL)

16.1 Introduction

Developers may extend the ZigBee application layer using any of the following techniques:

1. Private Profile: The profile ID is a two-byte value passed in ZigBee messages in the ZigBee APS frame. In order for two ZigBee devices to interact at the application layer, they must have the same profile ID. If they do not they will drop each others' messages. A private profile is used to completely protect all interaction within a given system. If you are planning to use ZigBee for your network and link layers but in other respects are planning to have a closed system, you may wish to create a private ZigBee Profile. If you use a private profile, your devices will not be interoperable with any other ZigBee devices using other profiles.
2. Manufacturer-Specific Clusters: Any clusters with cluster IDs of range 0xfc00 – 0xffff are considered manufacturer-specific and must have an associated two-byte manufacturer code. All commands and attributes within a manufacturer-specific cluster are also considered manufacturer-specific.

   Example: In the sample-extensions.xml file included with the application framework, we have defined a sample manufacturer-specific cluster with Cluster ID 0xfc00 and manufacturer code 0x1002 (Silicon Labs' manufacturer code).
3. Manufacturer-Specific Commands: You can augment a standard ZigBee cluster by adding manufacturer-specific commands to that cluster. Manufacturer-specific commands within a standard ZigBee cluster may use the entire range of command IDs 0x00 – 0xff. A two-byte manufacturing code must be provided for the manufacturer-specific command so that the command can be distinguished from the standard ZigBee commands in that cluster.

   Example: In the sample-extensions.xml file included with the application framework, we have defined three commands to extend the On/Off cluster called OffWithTransition, OnWithTransition and ToggleWithTransition. These commands share the same command IDs as the standard Off, On and Toggle commands in that cluster. However they also include the manufacturer code 0x1002, indicating that they are Silicon Labs' manufacturer-specific commands.
4. Manufacturer-Specific Attributes: Standard ZigBee clusters can be extended by adding manufacturer-specific attributes to your application. Manufacturer-specific attributes within a standard ZigBee cluster may use the entire attribute ID address space from 0x0000 to 0xffff. A two-byte manufacturer code must be included for each manufacturer-specific attribute so that it can be distinguished from non-manufacturer-specific attributes.

   Example: In the sample-extensions.xml file included with the application framework, we have defined a single attribute Transition Time which shares the same attribute ID with the on/off state in the on/off cluster 0x0000. However, the transition time attribute also contains the manufacturer code 0x1002, indicating that it is Silicon Labs' manufacturer-specific attribute.

Note: Silicon Labs' manufacturer code 0x1002 is defined by the ZigBee organization and is included in the Manufacturer Code database (ZigBee document #053874). Manufacturer codes are required for the implementation of manufacturer-specific clusters, attributes and commands. Unique manufacturer codes are provided by ZigBee for each requesting organization. To get a manufacturer code for your organization contact ZigBee at http://zigbee.org.

16.2 Limitations to Consider

There are two notable limitations to consider when extending the application framework with manufacturer specific clusters, attributes and commands.

- All cluster IDs including those of manufacturer-specific clusters MUST be unique within a single device. The Ember application framework does not currently support overlapping manufacturer-specific cluster IDs within a single device. In other words, you cannot implement cluster 0xFC00 with manufacturer code 0xFEED AND cluster 0xFC00 with manufacturer code 0xBEEF on the SAME device. The Ember application framework assumes that ALL cluster IDs are unique regardless of the manufacturer code associated with them.
- All attribute and command IDs within a manufacturer-specific cluster MUST be unique, and are assumed to have the same manufacturer code as the cluster they are in. The ZigBee protocol does not support overlapping manufacturer-specific attribute or command IDs (with different manufacturer codes) WITHIN a manufacturer-specific cluster. The reason is simply that only a single manufacturer code is passed in the ZigBee application header. If the cluster addressed is in the manufacturer-specific range 0xFC00 – 0xFFFF then the manufacturer code is assumed to apply to the cluster. This makes it impossible to address, for instance, Attribute 0x0000 with manufacturer code 0xFEED inside cluster 0x0000 with manufacturer code 0xBEEF. The Ember application framework does not even bother to store individual manufacturer codes for attributes within a manufacturer-specific cluster since the manufacturer code of the cluster is assumed to apply to all of the attributes within it.

16.3 Defining ZCL Extensions within the Application Framework and Ember AppBuilder

The entire ZigBee Cluster Library is defined in XML format in the tool/appbuilder directory. In addition to expected XML files such as general.xml or ha.xml that describe the clusters, commands and attributes associated with standard ZCL used by the Ember application framework, there is a sample extension file called, unsurprisingly, sample-extensions.xml. This XML file contains several sample ZigBee extensions including a custom cluster, custom attributes added to the on/off cluster and custom commands added to the on/off cluster.

In order to extend the ZigBee cluster library, you must create a similar extension file for your extensions and add them into the Ember AppBuilder by following the instructions included in document UG112, *Ember AppBuilder User's Guide*, and the Ember AppBuilder Online Help in the section entitled "Creating Custom Clusters."

Your custom XML configuration files can be validated using the XML Schema Definition file (XSD) for the Ember application framework and Ember AppBuilder located at tool/appbuilder/appbuilder.xsd. Further documentation about extending the Ember application framework is included in the sample-extensions.xml file.

Note: Any multi-byte numeric constant values specified in the XML file should specify the full number of digits as hex, such as "0x000000000000" (for an int48u) rather than simply "0x00" or "0". This ensures that the proper default value will be added to the GENERATED_DEFAULTS define in the <appname>_endpoint_config.h file during generation.

16.4 Manufacturer-Specific Attribute APIs

Some APIs in Ember application framework used to interact with attributes have been modified to take a manufacturer code as an argument.

16.4.1 Attribute Read and Write

All of the read and write attribute functions have additional functions that take a manufacturer code along with the rest of the attribute-addressing information. When you read and write to a manufacturer-specific attribute you must supply the manufacturer code for the attribute you wish to read or write so that it can be found in the attribute table. For example, you may read a standard ZigBee attribute using the function emberAfReadServerAttribute. However,

if you call this function for a manufacturer-specific attribute no manufacturer Code argument allows you to properly identify your manufacturer-specific attribute, so the read will fail. If you wish to read a manufacturer-specific attribute you must use the manufacturer-specific functions emberAfReadManufacturerSpecificServerAttribute and emberAfReadManufacturerSpecificClientAttribute. Both of these functions take a manufacturer code, which they pass on to the general function emberAfReadOrUpdateAttribute.

By breaking out the manufacturer-specific APIs into their own interface, we eliminate the need for code that is non manufacturer-specific to pass around bogus manufacturer codes. This would be a waste of code space given the large number of attribute interactions that exist in the application framework.

16.4.2 Attribute Changed Callbacks

We have also added manufacturer-specific attribute changed callbacks into the Ember application framework, so that standard attribute callbacks do not need to waste code space checking a bogus manufacturer code.

16.4.3 Sample Application

The SeMeterMirror sample application uses a Silicon Labs manufacturer-specific attribute to store the IEEE address of the mirrored device in persistent storage. This is a very convenient usage of the manufacturer-specific attribute functionality. By defining the attribute using Silicon Labs' manufacturer code 0x1002 we were able to leverage the Ember application framework's attribute storage mechanism and created a value that we may automatically interact with over the air as well as within our application. For more information on this specific example see the SeMeterMirror sample application located at app/framework/sample-apps/se-meter-mirror/SeMeterMirror_callbacks.c.

17 Designing an Application with Ember AppBuilder

Ember AppBuilder is a tool for generating ZigBee-compliant applications. Ember AppBuilder is made up of two parts: the Ember application framework and a graphical tool for configuring the included source code. The Ember AppBuilder graphical tool is both a stand-alone application and an Ember Desktop plug-in. Ember AppBuilder gives you an interface for turning on or off embedded clusters and features in the code compiled into a finished application.

Ember AppBuilder is intended to meet the following goals:

- Quickly create ZigBee-compliant applications for the EM250, EM260, and EM35x processors.
- Enable rapid development and decrease customer time-to-market by providing standard SE and HA applications.

17.1 ZCL Concepts

17.1.1 Definitions

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.

Ember AppBuilder currently supports three ZigBee application profiles:

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

Clusters

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.

The ZigBee Cluster Library (ZCL) is a document that specifies the 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.

Devices

A ZigBee device can be thought of as a collection of clusters. For example, an on/off light switch and an on/off light are two of the 31 devices in the HA 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 on the same network. If devices have different security settings, they cannot join together. If a user buys an HA device from company A and buys an HA device from company B, because they use the same application profile 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. Interoperability at the application level is not guaranteed until they use an application profile. These standard application profiles enable Ember AppBuilder to generate compliant ZigBee applications.

The HA on/off light has the following implementations:

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

The HA on/off light switch has 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.

17.1.2 More About Clusters and Attributes

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 ZCL 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 Light Switch sends “on”, “off”, and “toggle” commands that the Light can receive (and understand). It also defines that the Light stores a Boolean attribute called “on/off” representing the current state of the 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.

17.1.2.1 Example: The Identify Cluster

The client/server interaction defined by the ZCL is illustrated in the Identify example shown in Figure 9.

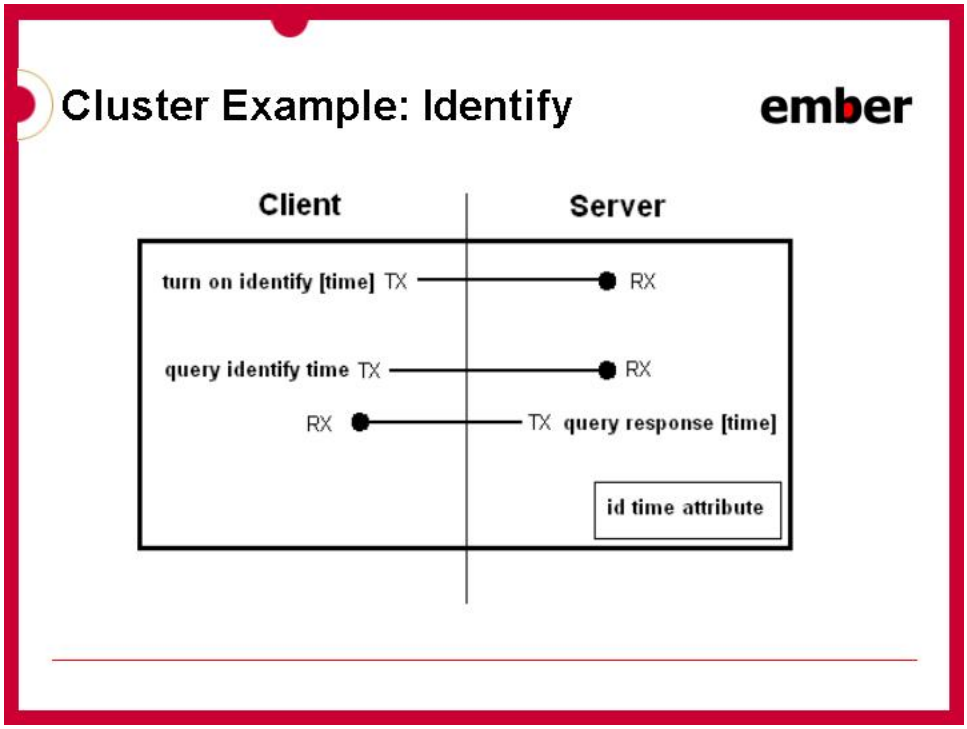

Figure 9. Cluster Example: Identify

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 is 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 A 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 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.

17.1.2.2 Example: The Temperature Measurement Cluster

Figure 10 illustrates another example of a cluster. This example shows temperature measurement.

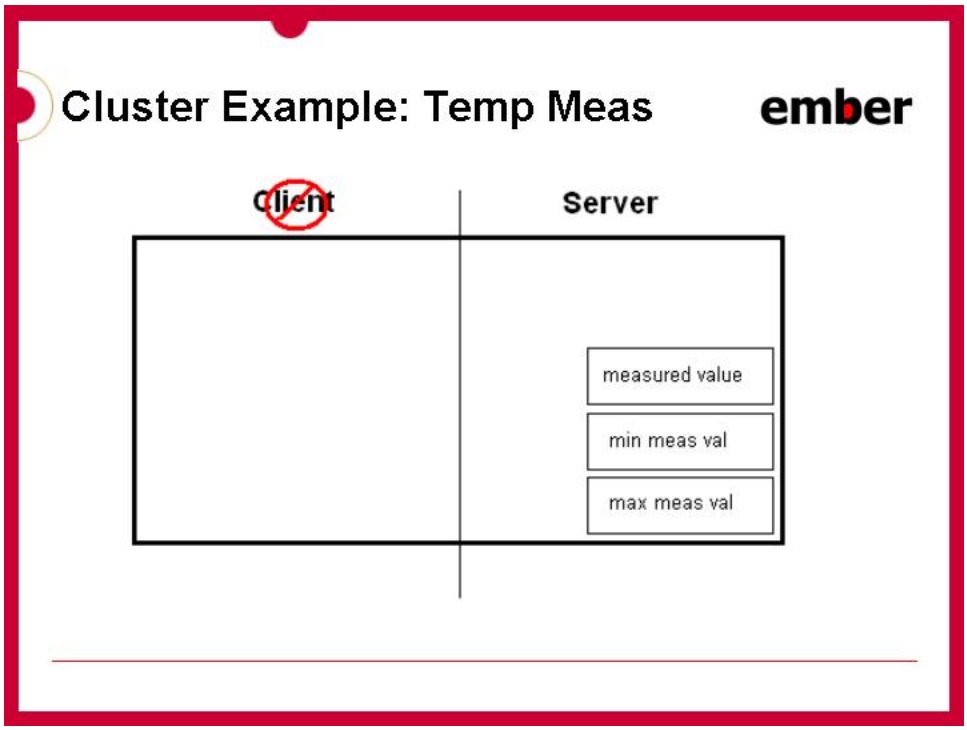

Figure 10. Cluster example: Temperature measurement

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.

Note: Fourteen global commands 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.

17.2 Ember AppBuilder and the Application Framework Architecture

Ember AppBuilder is a GUI tool which is part of the Ember Desktop application and is used to configure the Ember application framework code. Ember AppBuilder reads configuration data out of the installed stack directory. The .properties and .xml files located in tool/appbuilder tell Ember AppBuilder everything it needs to know about the associated stack. By interpreting these configuration files within the stack, Ember AppBuilder is able to generate the appropriate configuration data and project files necessary for a complete ZigBee application. Figure 11 shows how Ember AppBuilder works with the Ember application framework.

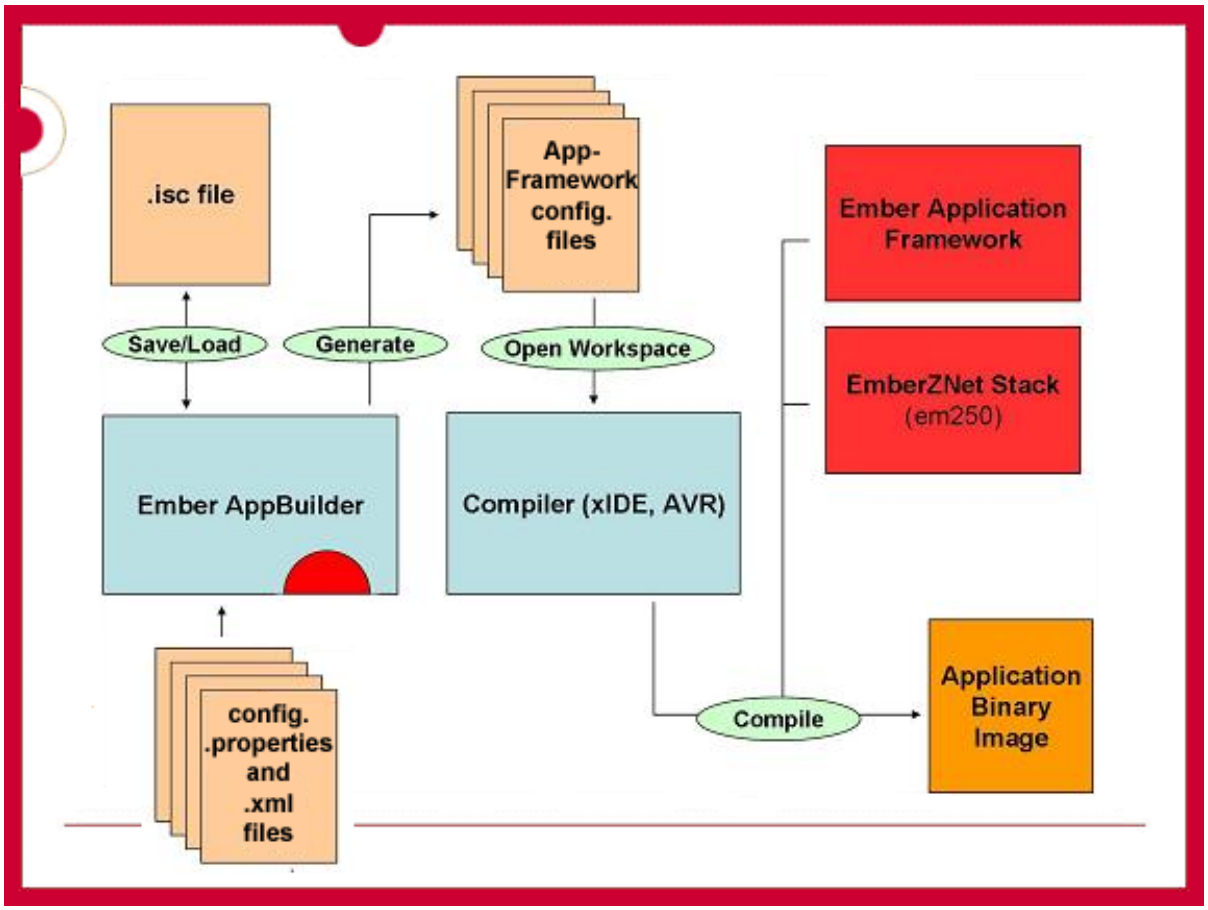

Figure 11. How Ember AppBuilder and the Application Framework Work

17.2.1 The Ember AppBuilder GUI

The GUI does not generate any code—all the code already exists. The code is protected with # define, as shown in Figure 12.

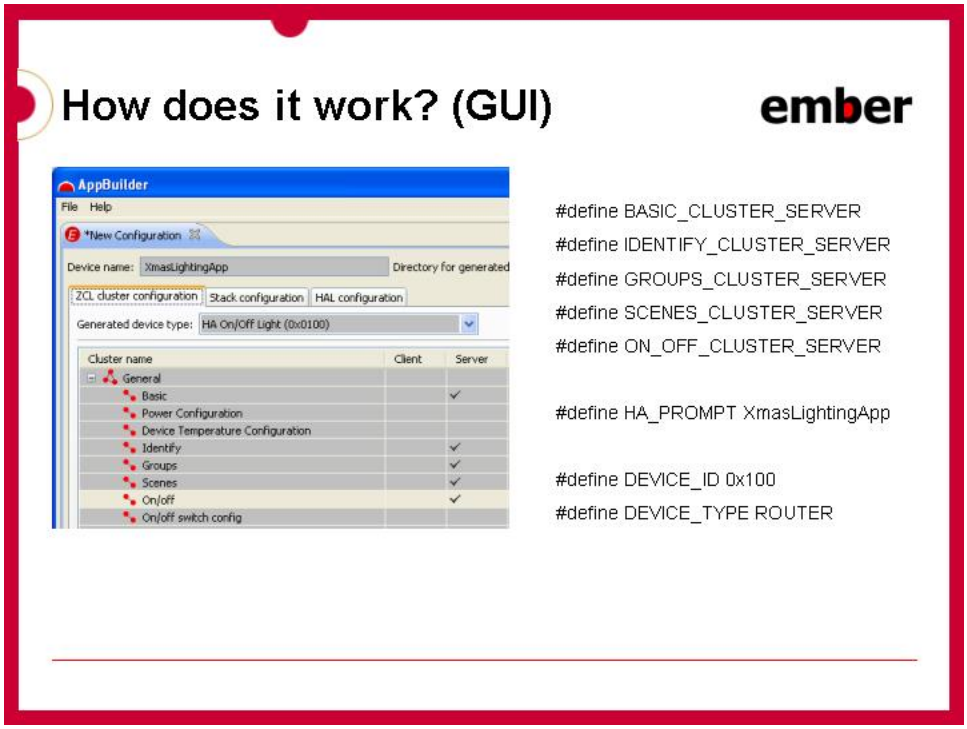

Figure 12. How does it work? (GUI)

In Figure 12, the Ember 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 configuration files generated by Ember AppBuilder. By using the # define, it is possible to turn on or off different features of the embedded code.

You can choose a pre-defined device from the GUI that specifies an established set of clusters, or you can choose your own custom device. Stack options are available for security, PAN ID, xPAN, and sleep time (that is, nap/hibernate). HAL options are available 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, you get 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 to determine when you have gone outside of your module if you try to build without a particular cluster, and another piece of the code is expecting that cluster to be there.

It is important to understand that you 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.

Sleep time includes a nap and a hibernate time. The nap time configuration makes it possible to send messages reliably to the device provided the nap time is shorter than the timeout used to retrieve messages from a sleepy device's parent. The hibernation configuration allows for a sleep interval that is longer than the indirect transmission timeout period.

You 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 you choose EM250, you can

choose the bootloader, the debug level, and the serial ports you want to use, and specify how to configure the GPIOs.

CONTACT INFORMATION

Silicon Laboratories Inc.

400 West Cesar Chavez
Austin, TX 78701
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032

Please visit the Silicon Labs Technical Support web page for ZigBee products:
www.silabs.com/zigbee-support and register to submit a technical support request