ember

EM35x

EM250

EM260

Application Framework V2 Developer Guide

20 May 2011

120-3028-000D

Ember Corporation
25 Thomson Place
Boston, MA 02210
+1 (617) 951-0200
www.ember.com

wireless semiconductor solutions

Contents

1 Introduction

1.1 Purpose

The Ember Application Framework is a body of embedded C code that can be configured by Ember's AppBuilder to implement any ZigBee Cluster Library (ZCL) application. Ember Application Framework Version 2 (AFV2) represents a significant update to Ember's previous framework code (V1). The V1 framework, located in the app/ha directory, will be deprecated. Application Framework Version 2 is located in the app/framework directory.

This guide covers the structure and usage of the Application Framework Version 2. Where appropriate, we have added information outlining differences between the latest release of the Application Framework and its predecessors.

1.2 Building an Application

An application is created in several steps using the Application Framework.

1. Create Application Framework configuration files using AppBuilder. The configuration files as well as the project files for your platform of choice are generated by AppBuilder. More information on how to use AppBuilder can be found in the 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 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 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 InSight Desktop (ISD) console window and the applications command line interface. More information on how to use InSight Desktop is available in the online help in ISD (Help | Help Contents).

1.3 New Features in AFV2

AFV2 includes the following customer-requested features:

1. **Multiple Endpoint Support**: An application may now implement many different devices on different endpoints from 1 – 239. (Endpoints 0 and 240 – 255 are reserved by the ZigBee specification). In V1 each device had only a single default endpoint.
2. **Application Framework Callbacks:** The callback API allows you to abstract your code away from the framework. In V1 application code was included directly into the framework, which made it difficult to migrate to a new framework release.
3. **External Attributes:** Attributes may be stored in a location external to the framework, and retrieved or updated by the framework through the callback interface. This means that you no longer need to replicate a device's attribute data within the framework's attribute storage. This feature provides RAM savings and simplifies state management.

4. **Singleton Attributes:** Attributes that can have a single value across all endpoints may be identified as "singleton". This means that, while each endpoint looks like it includes that attribute, in reality only a single attribute is stored within the framework. This feature provides RAM savings and simplifies state management.

5. **Attribute Persistence:** Attributes may also be persisted in SIMEEPROM when using an Ember System-on-Chip like the EM250 or EM35x family of chips. If you are not using one of these chips and instead are using a two-chip solution, you may indicate that the attribute in question is externally stored, and then manage its persistence through the host chip's method.

6. **Unified and Documented Application Framework API:** The Application Framework API has been modified across the board to use the 'emberAf' prefix and is referenced in app/framework/include/af.h. Items in the API include how to interact with the attribute storage, how to send and respond to a ZCL command, as well as other useful utilities. This feature brings the Application Framework API into line with the Ember stack API included in stack/include/ember.h.

7. **Custom Cluster Support:** You can now create your own custom clusters by creating a cluster .xml file and importing it into the framework through the AppBuilder preferences dialog. All of the ZCL-specific code used by the Application Framework is generated by AppBuilder into the app/framework/gen directory. For more information on how to create a custom cluster please see the online help within AppBuilder.

8. **Generated Command Handling:** ZigBee cluster commands are now (optionally) handled by generated code included in app/framework/gen/call-command-handler.c. This allows you to leverage the Application Framework's command handling for the handling of your own custom cluster commands. Optionally you can choose not to generate command handling code by including <generateCmdHandlers>false</generateCmdHandlers> in the XML description of their custom cluster. For an example see tool/appbuilder/ami.xml.

9. **Updated Command Line Interface (CLI):** The Application Framework CLI has been updated to include the following enhancements:

   - Uses less code space.
   - Includes error detection and usage feedback.
   - Uses deterministic parsing of values. Every command may include values in either hex or decimal. If a value includes the '0x' prefix, it is interpreted as a hex value; otherwise it is interpreted as decimal.
   - Handles strings and arrays of any size.

10. **Granular Debug Printing:** Application Framework V1 included only four options for debug printing. Application Framework V2 extends this so that debug printing can be included or excluded and turned on and off within each cluster and each general area in the framework. This feature saves on code, RAM, and visual clutter during development.

11. **Include User-Defined Code Files in the Generated Project File:** When users generated a project file from AppBuilder in Application Framework V1, it would overwrite their existing project file and any code files they had added in the Integrated Development Environment (IDE) would be lost. AppBuilder now includes a tab in which user-defined includes may be entered, which will in turn appear in the project file generated by AppBuilder.

12. **Application Framework Plugins:** Plugins provide an Ember implementation of the ZCL cluster code. All of the code that used to be located in app/framework/cluster has been moved into individual plugins in app/framework/plugin. This gives you the option of including or excluding the Ember-created cluster code from your application. The plugins generally provide useful implementations of ZCL command handling functions. If you choose not to include a plugin, you must implement the cluster callbacks for all required commands within a cluster, and add your own code to handle them.

13. **Application Events:** Ember's plugins now use the Ember event mechanisms to schedule and execute application functions. The event mechanism centralizes event tracking within the system. This centralization allows the device to query the event table before it sleeps so that it knows exactly how long it can sleep. In addition, events can be scheduled to keep the processor out of hibernation as long as the event is scheduled. This feature is useful for

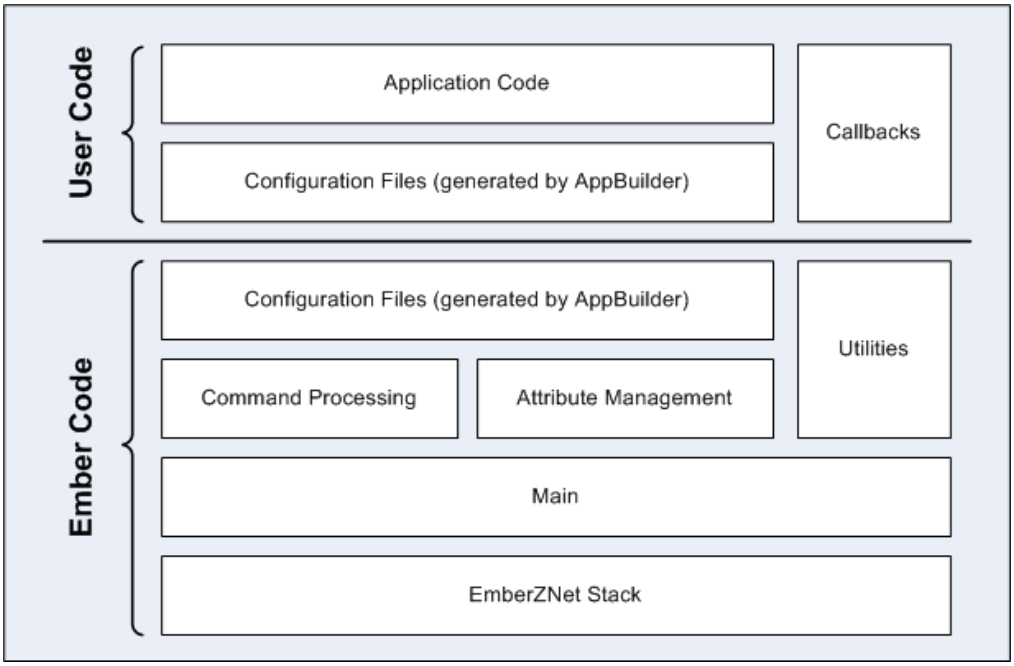
timeout-type events which expect the device to be awake and doing something important, until the event is called indicating a timeout.

2 Application Framework Architecture

The 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 Application Framework is the separation of user-created and Ember-created code. While Ember provides all of the source code for the Application Framework, user-created code should live outside the framework and should interact with the framework through the Application Framework API exposed by the framework utilities and callbacks. The block diagram in Figure 2-1 shows a high level overview of the Application Framework architecture and how the two code bases are separated.

Figure 2-1. Application Framework Architecture



The main file included in app/framework/util consumes the Ember Stack handler interface and ties the Application Framework into the EmberZNet 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 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 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 located in app/framework/gen. This is important for creating custom clusters. The process of creating a custom cluster and generating the appropriate handlers and command processing code is described in the AppBuilder documentation located within AppBuilder under Help | Help Contents.

3 Application Framework Directory Structure

tool/appbuilder: Configuration and template files used by AppBuilder
When you point 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 AppBuilder Help at Help | Help Contents | Creating custom clusters

app/framework: All of the Application Framework V2 code is located in app/framework. Instead of having all the code in a single directory (as was the case with V1), the major portions of the code have been broken out into their own directories

app/framework/cli: Code related to AFV2's implementation of Ember's upgraded CLI
Core code for the new CLI is included in app/util/serial/command-interpreter2.c. The new CLI adds data type checking and command usage feedback among other things. As a result:

1. All commands now 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/framework/gen: All framework code generated from the XML files located in tool/appbuilder
The XML description of the ZCL must at some point be translated into a set of header and, in some cases, .c files. The code in app/framework/gen IS NOT generated on the fly when you click on the main GENERATE button for your configuration. It is pre-generated by Ember during the release process and included as static files in the framework release. You can regenerate these files at any time in AppBuilder's preference dialog. For more information, see AppBuilder help at Help | Help Contents | Creating custom clusters.

For the most part, the files in this directory are simply C headers used by the framework to interact with the ZCL, with one notable exception. The call-command-handler.c file implements the generated command handling for the ZCL's cluster specific commands. Incoming cluster-specific commands are parsed by this code and sent directly to a cluster callback, which may be optionally implemented by the user.

app/builder: The output location for all generated files from the AppBuilder
When you generate an application from AppBuilder, it puts the generated files into this directory under a directory of the same name as the device name within the 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 AppBuilder. AppBuilder automatically copies the sample application files into the associated directory within app/builder.

app/framework/include: All of the external APIs for the Application Framework
This directory mirrors the use of the include directory in the stack. It is intended to be the single location for all application interfaces. Anything that that the application uses from the framework is exposed through an API documented in this directory.

app/framework/plugin (new to AFV2 Beta III): All Ember-created ZCL cluster code
This directory contains all of the cluster code created by the Ember team for handling cluster commands. This code optionally can be included in an application by selecting the plugin from 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/report: All framework code related to reporting
The code includes Ember's support for reporting within the framework. Please note that Ember's proprietary Out of the Box reporting has been removed from the Application Framework V2 GA.

app/framework/sample-apps: All sample applications which use the application framework
These sample applications may be opened within AppBuilder. 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. In V1 these files were called ami-security and ha-security.

app/framework/util: The application's mains, message processing, and any other utility code used by the Application Framework
This directory contains the guts of the Application Framework V2. In V1, this code was included in files with the zcl-util prefix. The new 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 Application Framework used a single header file to configure the 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, AppBuilder also generates an "endpoint configuration" header file with the suffix endpoint_configuration.h.

<DeviceName>_endpoint_configuration.h
The generated file that configures the 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 AppBuilder. Ember 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.

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

<DeviceName>.ewp, eww, .xip, .xiw, .mak
Generated project files for your application. 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.

5 The Application Framework API

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

Many of the functions in Application Framework now 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` has moved from app/ha/zcl-util-attribute-table.c to app/framework/util/attribute-table.c, and the signature of the function has been changed to take the endpointId as its first argument.

Some examples of the Application Framework include:

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

All of the Application Framework APIs intended to be used by the customer application now include the "emberAf" prefix. This includes all functions that previously included the "zclUtil" (V1) or "af" prefix.

APIs for getting information about endpoints and attributes have been added 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 now 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 Application Framework callbacks are intended to be used as a means to remove all customer code from the Application Framework. If any of your application code needs to be put into the Application Framework, Ember views this as a bug with the 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 http://www.ember.com/support_index.html.

Generally, when a callback is called the 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 Application Framework's handling of a particular message, return TRUE to indicate that the message was complete. This ensures that the Application Framework does not interfere with your handling of the message.

6.1 Callback Generation

AppBuilder has the ability to generate a stub callback file for you. By default, 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, AppBuilder protects your generated callbacks file from being overwritten by asking if you want to overwrite it. By default, AppBuilder will not overwrite any previously created callbacks file. If you choose to overwrite the file, 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 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 Application Framework code in locations where customers have indicated that they wish to receive information about the function of the Application Framework.

All global commands fall into this category. The 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 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 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 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 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.

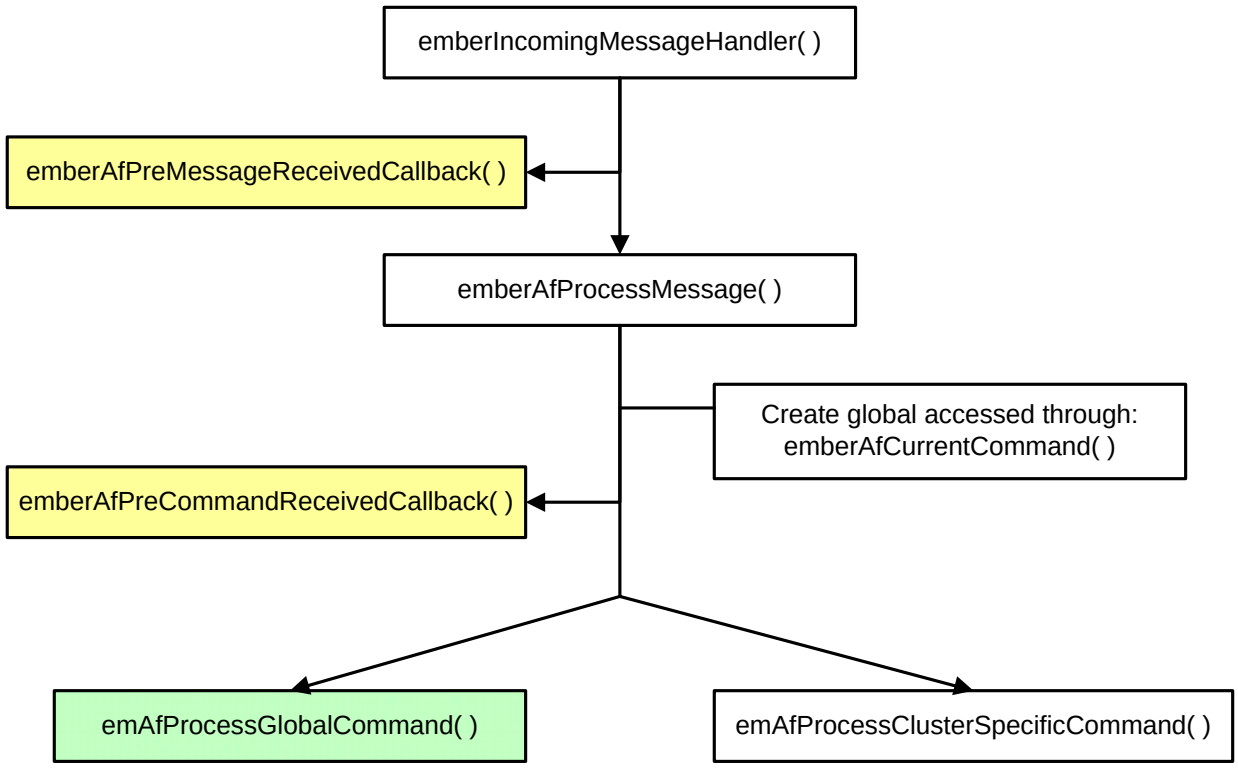
6.3.3 Global command callbacks

ZigBee global commands are also covered in the 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. The global command callbacks are covered in sections 6.4 through 6.10 below.

6.4 Callback Flow

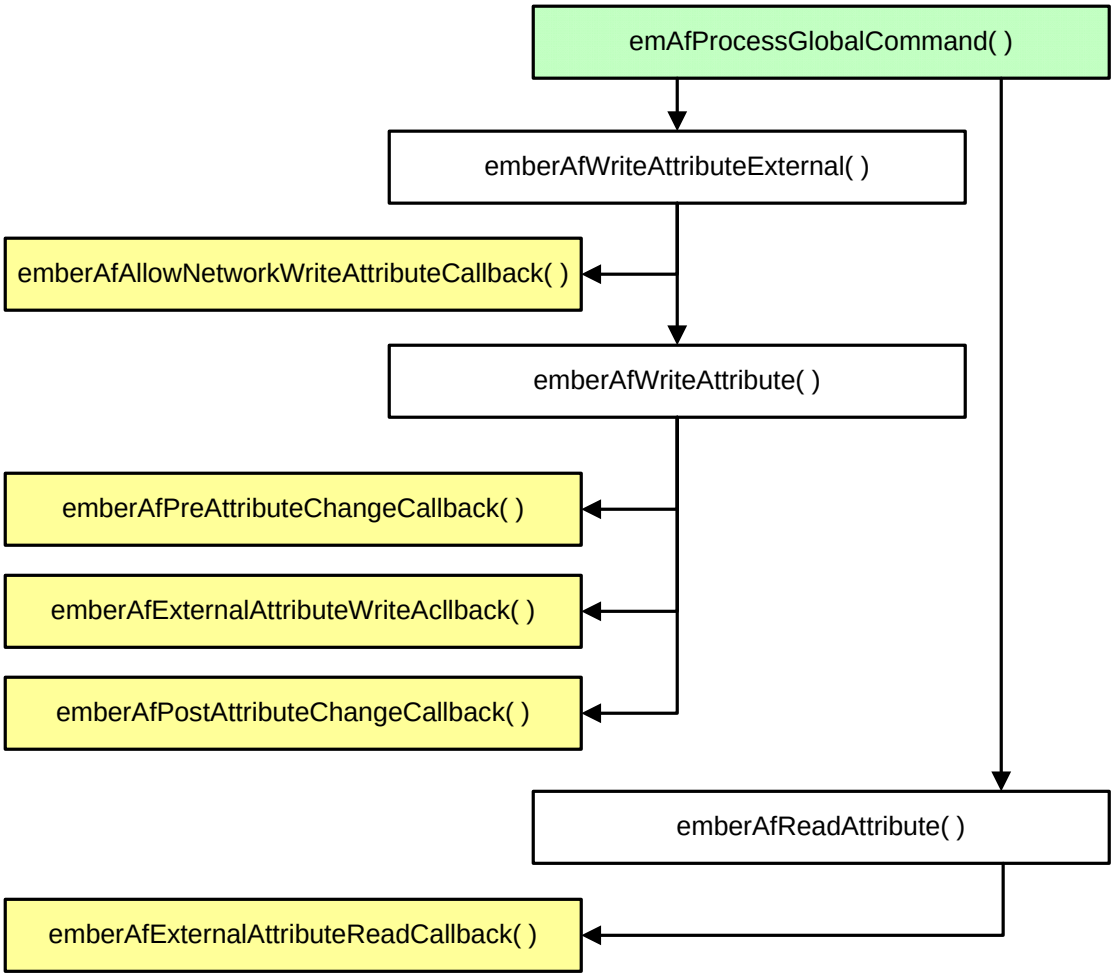
Figure 6-1 shows how a message received by the application framework's implementation of emberIncomingMessageHandler is processed and flows through the framework code and out to the application implemented callbacks.

Figure 6-1. Incoming Message Flow



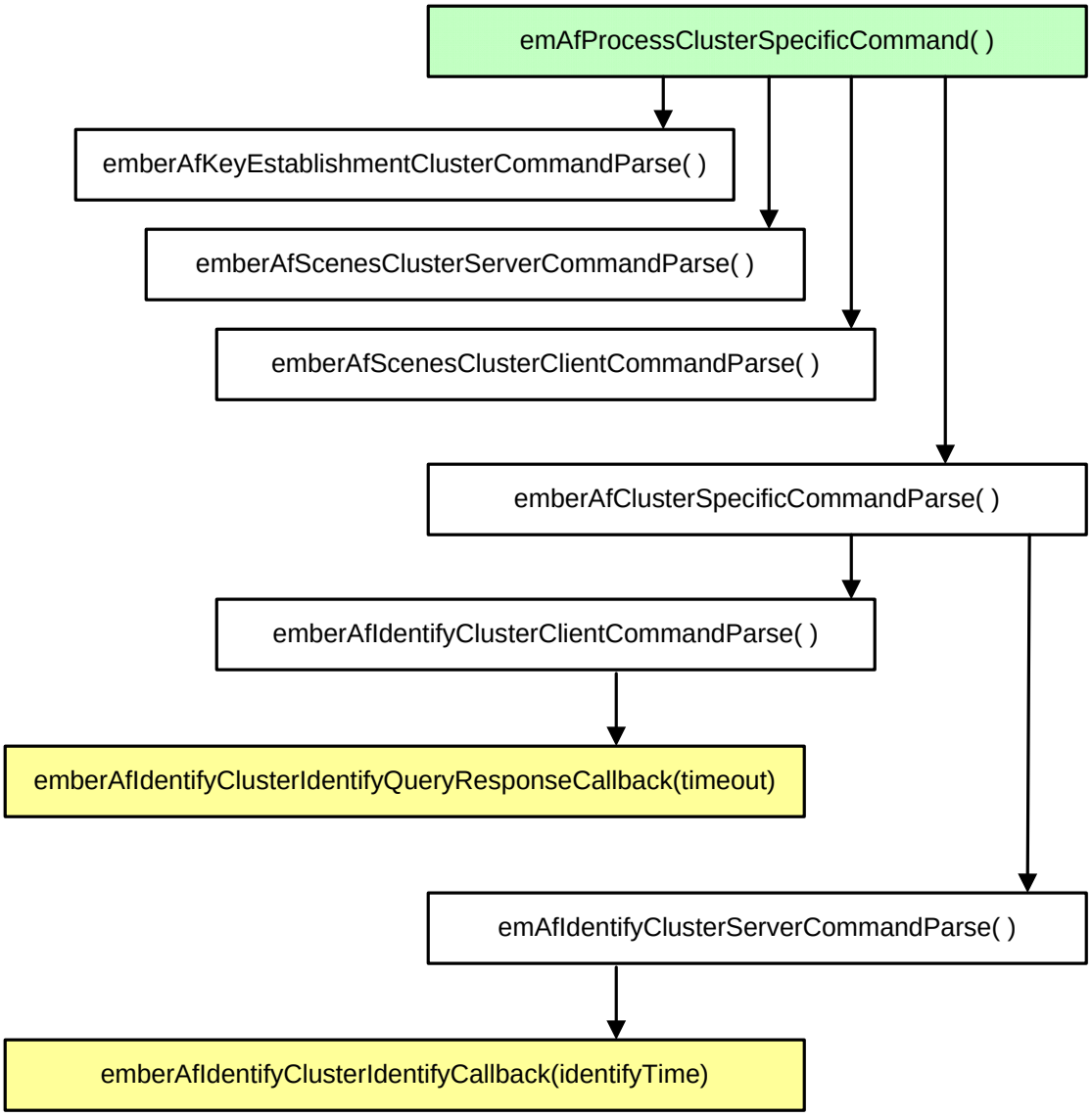
Once the incoming message is determined to be an incoming global command, it is passed off to the global command handling for processing, as shown in Figure 6-2.

Figure 6-2. 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 6-3.

Figure 6-3. Cluster-Specific Command Processing



6.5 emberAfReadAttributesResponseCallback

This function is called by the Application Framework when a Read Attributes Response command is received from an external device. The application should return TRUE if the message was processed or FALSE if it was not.

```
boolean emberAfReadAttributesResponseCallback(EmberAfClusterId clusterId,
                                              int8u * buffer,
                                              int8u bufLen) {
  return FALSE;
}
```

6.6 emberAfWriteAttributesResponseCallback

This function is called by the Application Framework when a Write Attributes Response command is received from an external device. The application should return TRUE if the message was processed or FALSE if it was not.

```
boolean emberAfWriteAttributesResponseCallback(EmberAfClusterId clusterId,
                                               int8u * buffer,
                                               int8u bufLen) {
  return FALSE;
}
```

6.7 emberAfConfigureReportingResponseCallback

This function is called by the Application Framework when a Configure Reporting Response command is received from an external device. The application should return TRUE if the message was processed or FALSE if it was not.

```
boolean emberAfConfigureReportingResponseCallback(EmberAfClusterId clusterId,
                                                  int8u * buffer,
                                                  int8u bufLen) {
  return FALSE;
}
```

6.8 emberAfReadReportingConfigurationResponseCallback

This function is called by the Application Framework when a Read Reporting Configuration Response command is received from an external device. The application should return TRUE if the message was processed or FALSE if it was not.

```
boolean emberAfReadReportingConfigurationResponseCallback(EmberAfClusterId clusterId,
                                                          int8u * buffer,
                                                          int8u bufLen) {
  return FALSE;
}
```

6.9 emberAfReportAttributesCallback

This function is called by the Application Framework when a Report Attributes command is received from an external device. The application should return TRUE if the message was processed or FALSE if it was not.

```
boolean emberAfReportAttributesCallback(EmberAfClusterId clusterId,
                                        int8u * buffer,
                                        int8u bufLen) {
  return FALSE;
}
```

6.10 emberAfDefaultResponseCallback

This function is called when the Application Framework receives a default response message over the network. The application should return TRUE if the default response was handled and FALSE otherwise so that it may be handled by the Application Framework.

```
boolean emberAfDefaultResponseCallback(EmberAfClusterId clusterId,
                                       int8u commandId,
                                       EmberAfStatus status) {
  return FALSE;
}
```

6.11 emberAfDiscoverAttributesResponseCallback

This function is called by the Application Framework when a Discover Attributes Response command is received from an external device. The Discover Attributes Response command contains a Boolean indicating if discovery is complete and a list of zero or more attribute identifier/type records. The application should return TRUE if the message was processed or FALSE if it was not.

```
boolean emberAfDiscoverAttributesResponseCallback(EmberAfClusterId clusterId,
                                                  boolean discoveryComplete,
                                                  int8u * buffer,
                                                  int8u bufLen) {
  return FALSE;
}
```

6.12 emberAfAllowNetworkWriteAttributeCallback

This function is called whenever a network device attempts to write a local attribute. With it you can implement permission security for network devices writing attributes over the air using the general write attributes command. In addition, this function may return a value indicating that an attribute that is commonly "read only" MAY be written by the calling device. This is important for circumventing the read-only nature of attributes where an external device must set those attributes.

Example: In mirroring simple metering data on an Energy Services Interface (ESI) (formerly called Energy Service Portal (ESP) in SE 1.0).), a mirrored simple meter needs to write read-only attributes on its mirror. The-meter-mirror sample application, located in app/framework/sample-apps, uses this callback to allow the mirrored device to write simple metering attributes on the mirror regardless of the fact that most simple metering attributes are defined as read-only by the ZigBee specification.

Note: The ZCL specification does not (as of this writing) specify any permission-level security for writing writeable attributes. As far as the ZCL specification is concerned, if an attribute is writeable, any device that has a link key for the device should be able to write that attribute. Furthermore if an attribute is read only, it should not be written over the air. Thus, if you implement permissions for writing attributes as a feature, you MAY be operating outside the specification. This is unlikely to be a problem for writing read-only attributes, but it may be a problem for attributes that are writeable according to the specification but restricted by the application implementing this callback.

```
void emberAfAllowNetworkWriteAttributeCallback(int8u endpoint,
                                               EmberAfClusterId clusterId,
                                               EmberAfAttributeId attributeId,
                                               int8u* value) {

}
```

6.13 emberAfPreAttributeChangeCallback

This function is called directly before an attribute is written into the attribute storage within the framework. It says that an attribute is about to change, and provides the new attribute value. Use this callback when you need to do something before an attribute is written into the framework.

```
void emberAfPreAttributeChangeCallback(int8u endpoint,
                                       EmberAfClusterId clusterId,
                                       EmberAfAttributeId attributeId,
                                       int8u type,
                                       int8u size,
                                       int8u* value) {

}
```

6.14 emberAfPostAttributeChangeCallback

This function is called after an attribute has been changed, and can be used as a notifier that an attribute has been changed. You can use this callback in order to react to attribute changes within your application.

```
void emberAfPostAttributeChangeCallback(int8u endpoint,
                                        EmberAfClusterId clusterId,
                                        EmberAfAttributeId attributeId,
                                        int8u type,
                                        int8u size,
                                        int8u* value) {

}
```

6.15 emberAfPreMessageReceivedCallback

This callback is the first in the Application Framework's message processing chain. The Application Framework calls it when a message has been received over the air but has not yet been parsed by the ZCL command-handling code. If you wish to parse some messages that are completely outside the ZCL specification or are not handled by the Application Framework's command handling code, you should intercept them for parsing in this callback.

This callback returns a Boolean value indicating whether or not the message has been handled. If the callback returns a value of TRUE, then the Application Framework assumes that the message has been handled and it does nothing else with it. If the callback returns a value of FALSE, then the application framework continues to process the message as it would with any incoming message.

Note: This callback receives a pointer to an incoming message struct. This struct allows the application framework to provide a unified interface between both Host devices, which receive their message through the ezspIncomingMessageHandler, and SoC devices, which receive their message through emberIncomingMessageHandler.

```
boolean emberAfPreMessageReceivedCallback(EmberAfIncomingMessage* incomingMessage) {
 return FALSE;
}
```

6.16 emberAfPreCommandReceivedCallback

This callback is the second in the Application Framework's message processing chain. At this point in the processing of incoming over-the-air messages, the application has determined that the incoming message is a ZCL command. It parses enough of the message to populate an EmberAfClusterCommand struct. The Application Framework defines this struct value in a local scope to the command processing but also makes it available through a global pointer called emberAfCurrentCommand, in app/framework/util/util.c. When command processing is complete, this pointer is cleared.

```
boolean emberAfPreCommandReceivedCallback(EmberAfClusterCommand* cmd,
                                          boolean isInterpan) {
  return FALSE;
}
```

6.17 emberAfMessageSentCallback

This function is called by the Application Framework from the message sent handler, when it is informed by the stack regarding the message sent status. All of the values passed to the emberMessageSentHandler are passed on to this callback. This provides an opportunity for the application to verify that its message has been sent successfully and take the appropriate action. This callback returns a Boolean value of TRUE or FALSE. A value of TRUE indicates that the message sent notification has been handled and should not be handled by the Application Framework.

```
boolean emberAfMessageSentCallback(EmberOutgoingMessageType type,
                                   int16u indexOrDestination,
                                   EmberApsFrame* apsFrame,
                                   int8u msgLen,
                                   int8u* message,
                                   EmberStatus status) {
}
```

6.18 emberAfMainInitCallback

This function is called from the application's main function. It gives the application a chance to do any initialization required at system startup. Any code that you would normally put into the top of the application's main() routine should be put into this function.

Note: No callback in the Application Framework is associated with resource cleanup. If you are implementing your application on a Unix host where resource cleanup is a consideration, we expect that you will use the standard Posix system calls, including the use of atexit() and handlers for signals such as SIGTERM, SIGINT, SIGCHLD, SIGPIPE and so on. If you use the signal() function to register your signal handler, please mind the returned value which may be an Application Framework function. If the return value is non-null, please make sure that you call the returned function from your handler to avoid negating the resource cleanup of the Application Framework itself.

```
void emberAfMainInitCallback(void) {

}
```

6.19 emberAfClusterInitCallback

This function is called once for each cluster implemented on the device. If you need to do any cluster-related initialization, implement it in this function.

```
void emberAfClusterInitCallback(int8u endpoint,
                                int16u clusterId) {

}
```

6.20 emberAfPreGoToSleepCallback

This function is called directly before the device goes to sleep. The pre-go-to-sleep callback gives the application a chance to control whether the device is allowed to go to sleep for the given sleep duration. A return value of TRUE indicates that the application has overridden the sleep attempt and the device should not go to sleep. A return value of FALSE indicates that the application does not have a preference regarding sleep and the device may continue to go into sleep mode as expected.

```
boolean emberAfPreGoToSleepCallback(int32u sleepDurationAttempt) {
  return FALSE;
}
```

6.21 emberAfPostWakeUpCallback

This function is called after the device wakes up from sleep. It provides the amount of time the device was sleeping. If the application needs to service some routine after sleep has taken place, the associated code should be placed in this function.

```
void emberAfPostWakeUpCallback(int32u sleepDuration) {
}
```

6.22 emberAfMainTickCallback

This function is called from within the main tick. Any code that would have previously been placed in the `appTick()` function should be placed here. The main tick may be used to service any state machines within the application. For instance, if your application has a state such as joined or commissioned that needs to be tracked and maintained, do it in this callback as you would in the main of your application.

```
void mainTickCallback(void) {
}
```

6.23 emberAfExternalAttributeWriteCallback

This function is called whenever the Application Framework needs to write an attribute which is not stored within the data structures of the Application Framework itself. One of the new features in Version 2 is the ability to store attributes outside the Framework. This is particularly useful for attributes that do not need to be stored because they can be read off the hardware when they are needed, or are stored in some central location used by many modules within the system. In this case, you can indicate that the attribute is stored externally. When the framework needs to write an external attribute, it makes a call to this callback.

This callback is very useful for host micros which need to store attributes in persistent memory. Because each host micro (used with an Ember NCP like the EM260) has its own type of persistent memory storage, the Application Framework does not include the ability to mark attributes as stored in flash the way that it does for Ember's SoCs like the EM35x. On a host micro, any attributes that need to be stored in persistent memory should be marked as external and accessed through the external read and write callbacks. Any host code associated with the persistent storage should be implemented within this callback.

All of the important information about the attribute itself is passed as a pointer to an EmberAfAttributeMetadata struct, which is stored within the application and used to manage the attribute. A complete description of the EmberAfAttributeMetadata struct is provided in app/framework/include/af-types.h.

This function assumes that the application is able to write the attribute and return immediately. Any attributes that require a state machine for reading and writing are not candidates for externalization at the present time. The Application Framework does not currently include a state machine for reading or writing attributes that must take place across a series of application ticks. Attributes that cannot be written immediately should be stored within the Application Framework and updated occasionally by the application code from within the emberAfMainTickCallback.

If the application was successfully able to write the attribute, it returns a value of TRUE. A return value of FALSE indicates that the application was not able to write the attribute.

```
boolean emberAfExternalAttributeWriteCallback(int8u endpoint,
                                              EmberAfClusterId clusterId,
                                              EmberAfAttributeMetadata * attributeMetadata,
                                              int8u * buffer) {
  return FALSE;
}
```

6.24 emberAfExternalAttributeReadCallback

Like emberAfExternalAttributeWriteCallback above, this function is called when the framework needs to read an attribute that is not stored within the Application Framework's data structures.

All of the important information about the attribute itself is passed as a pointer to an EmberAfAttributeMetadata struct, which is stored within the application and used to manage the attribute. A complete description of the EmberAfAttributeMetadata struct is provided in app/framework/include/af-types.h

This function assumes that the application is able to read the attribute, write it into the passed buffer, and return immediately. Any attributes that require a state machine for reading and writing are not really candidates for externalization at the present time. The Application Framework does not currently include a state machine for reading or writing attributes that must take place across a series of application ticks. Attributes that cannot be read in a timely

manner should be stored within the Application Framework and updated occasionally by the application code from within the emberAfMainTickCallback.

If the application was successfully able to read the attribute and write it into the passed buffer, it should return a value of TRUE. A return value of FALSE indicates that the application was not able to read the requested attribute.

```
boolean emberAfExternalAttributeReadCallback(int8u endpoint,
                                             EmberAfClusterId clusterId,
                                             EmberAfAttributeMetadata * attributeMetadata,
                                             int8u * buffer) {
  return FALSE;
}
```

6.25 emberAfTrustCenterJoinCallback

This function is called from within the trust center join handler whenever a device joins the network and the trust center join handler is notified. The same arguments that are passed to the trust center join handler are passed to this callback. If you are implementing a trust center and you want to know about each device that joins your network, implement service code inside this callback. This callback is only a notification of the join action taken. It does not provide an opportunity for the application to modify or affect that decision.

```
void emberAfTrustCenterJoinCallback(EmberNodeId newNodeId,
                                    EmberEUI64 newNodeEui64,
                                    EmberNodeId parentOfNewNode,
                                    EmberDeviceUpdate status,
                                    EmberJoinDecision decision) {
}
```

6.26 emberAfGetCurrentTimeCallback

This function is called whenever the Application Framework needs to get the current real time within the system. It can be used to give the framework access to accurate time provided by a real time clock, for example. If this function does not have access to the accurate real time, it returns 0. For more information on how time is handled within the Application Framework, see Chapter 7, Time Handling.

```
int32u emberAfGetCurrentTimeCallback(void) {
  return 0;
}
```

6.27 emberAfSetTimeCallback

Like emberAfGetCurrentTimeCallback above, this function is called whenever the framework needs to set the real time on the system. Implement this callback if the device has access to a real time clock and has the ability to set the real time on the clock.

```
void emberAfSetTimeCallback(int8u year,
                            int8u month,
                            int8u day,
                            int8u hour,
                            int8u min,
                            int8u sec) {
}
```

6.28 emberAfRegistrationCallback

This callback is called when the device joins a network and the process of registration is complete. It provides a value of TRUE if the registration process was successful and a value of FALSE if registration failed.

```
void emberAfRegistrationCallback(boolean success) {
}
```

6.29 emberAfCustomCommandLineCallback

If you enable this callback, it will be called when 'custom' CLI command is executed. It is responsible for reading its own arguments using the command line parsing API, located in app/util/serial/command-interpreter2.h.

```
void emberAfCustomCommandLineCallback(void) {}
```

7 Time Handling

The 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/cluster/time.c.

The 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 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 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 Application Framework has two types of events: custom events and cluster events. Custom events are created by the Application Framework user and can be used for any purpose within the application. Cluster events are specifically related to the cluster implementations in the Application Framework's plugins.

8.1 Creating a Custom Event

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

The Application Framework and AppBuilder provide a helpful interface for creating and adding custom events to your application. To create a custom event in AppBuilder, open the "Includes" tab in your 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 Application Framework, as well as stubs for your custom event to the "callbacks" file generated by 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. 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 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 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 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: (New in Version 2 GA) In Application Framework V2 GA 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 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 Application Framework V2, 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 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 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 HAL Ember provides 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 Application Framework is written to run on any platform the user desires. As a result, the 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 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 Application Framework.

Mark an attribute as externally located by clicking on the "E" checkbox next to the attribute in the AppBuilder GUI. The attribute's metadata will be tagged to indicate that the Application Framework should not reserve memory for the storage of that attribute. Instead, when that attribute is to be read or written, the 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, Ember suggests that you include Application Framework storage and update the value in the 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 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 Application Framework and 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 AppBuilder. As a convenience, the 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 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.

9.1.7 Attribute Reporting (R) (Removed in Version 2 GA)

Ember's Out-Of-the-Box (OOB) reporting allows you to quickly set up a device to collect and report on attribute values. If OOB reporting is turned on, attributes that are marked for reporting are reported to any collector that requests them.

Note: Ember's OOB reporting is a manufacturer-specific feature created by Ember for the convenience of our customers. If you wish to use reporting inside the ZigBee specification, you should set up reporting manually using the reporting-related general commands.

9.2 Interacting with ZCL Attributes in Version 2

The Application Framework V2 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 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.

10 Command Handling and Generation

10.1 Sending Commands and Command Responses

The Application Framework API includes many useful macros for sending and responding to ZCL commands. All of the macros are defined in app/framework/gen/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 setting the global emberAfResponseWaitingFlag flag to TRUE.

   ```
   emberAfResponseWaitingFlag = TRUE;
   ```

10.2 ZCL Command Processing

When the 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 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 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 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.

Unlike V1, which had hard-coded expectations of how arguments were passed, the Application Framework V2 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 CLI has a callback that can be used to extend the CLI. When the emberAfCustomCommandLineCallback is implemented, any CLI command that starts with the term “custom” is passed to the callback for processing. Arguments passed on the command line can be read by using the command interpreter’s API located in app/util/serial/command-interpreter2.h. An example of how this is done is included in the Application Framework’s CLI located at app/framework/cli/zcl-cli.c.

For example, the specific API used to read integer commands off the command line is:

```
/** Retrieves unsigned integer arguments. */
int32u emberUnsignedCommandArgument(int8u index);

/** Retrieves signed integer arguments. */
int16s emberSignedCommandArgument(int8u index);
```

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 InSight 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 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 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

This section is a reference guide for bundled CLI commands. All of these commands are conditionally included in any application created using AppBuilder and the Application Framework, depending on the type of device created. Information about which types of devices support which commands is included.

11.3.1 Three Modes of the CLI

The Application Framework supports three different serial modes, Full, Minimal, and Tiny, which can be configured from AppBuilder.

1. FULL - Supports all the commands listed in this reference that pertain to the device. This mode uses the most flash. It is enabled by defining "ZA_CLI_FULL" and not defining "ZA_TINY_SERIAL_INPUT". This is the default configuration for applications created from AppBuilder.

2. MINIMAL - Supports a subset of the commands listed in this reference. It uses less flash than FULL mode and more flash than TINY mode. This mode is enabled by defining "ZA_CLI_MINIMAL" and not defining "ZA_TINY_SERIAL_INPUT"
    - reset
    - info
    - print
    - network
    - keys
    - raw
    - send
    - bsend
3. TINY - This mode saves the most flash and saves RAM over the MINIMAL and FULL modes. It supports only single character commands and does not provide all the functionality that the FULL CLI provides. It is enabled by defining "ZA_TINY_SERIAL_INPUT".
    - i = gives information about the local node.
    - f = forms a network on channel TINY_SERIAL_CHANNEL(11) and panID TINY_SERIAL_PANID (0x00AB).
    - j = attempts to join a network on channel TINY_SERIAL_CHANNEL(11) and panID TINY_SERIAL_PANID (0x00AB).
    - p = permits join for 20 seconds.
    - l = leaves network.
    - t = sends a read time cluster time attribute command to the ZigBee Coordinator (ZC).
    - b = sends a read basic cluster version attribute command to the ZC.
    - k = prints security keys.
    - c = clears security keys.

11.3.2 Command argument limit

By default, the CLI can only support up to 11 total command arguments. This is sufficient for most commands. However, if you wish to create a custom CLI command that requires more command arguments, you may wish to increase this limit in the CLI header file located at app/util/serial/command-interpreter2.h.

For example, by default the following command works because it has only 11 arguments:

```
zcl groups get 7 0001 0002 0003 0004 0005  0006 0007
```

However this one will not since it has 12:

```
zcl groups get 8 0001 0002 0003 0004 0005  0006 0007 0008
```

11.3.3 General Commands

- **help** - Shows the available commands.
- **version** - Shows the version of the software.
- **reset** - Resets the device.
- **counters** – Prints out the stack counters on the SOC if application debug printing is turned on. Stack counter indexes are documented in stack/include/ember-types.h in the EmberCounterType enum.
- **option button0** - Simulates button0 press.
- **option button1** - Simulates button1 press.
- **write** [endpoint in decimal or 0x<hex>] [cluster in decimal or 0x<hex>] [attrID in decimal or 0x<hex>] [mask in decimal or hex (0 for client or 1 for server)] [dataType in decimal or 0x<hex>] [all attribute value bytes as "string" or {hex}]- Writes the value specified to the attribute specified in the local attribute table. Example: `write 0x01 0x00 0x0010 0x21 { aa bb }`

11.3.4 Informational Commands

- **info** - Gives information about the local node. Application printing must be enabled by the define EMBER_AF_PRINT_APP. If application printing is not enabled, the CLI executes the command but nothing is printed to the command line.
- **option binding-table print** - Prints the binding table. Application printing must be enabled by the define EMBER_AF_PRINT_APP. If application printing is not enabled, the CLI executes the command but nothing is printed to the command line.
- **option binding-table clear** - Clears the binding table.
- **option binding-table set** [1 byte binding table index] [2 byte cluster id] [1 byte local endpoint] [1 byte remote endpoint] [8 byte IEEE address provided big endian] - Sets a binding table entry for the arguments specified.

11.3.5 Debug Printing commands

Note: Some printing commands require that debug printing is turned on for the associated cluster. Since the CLI commands use the emberAfxxxPrint commands, if debug printing is not turned on for the cluster in question nothing will print on the command line.

If debug printing is compiled into an application but not turned on by default, it can be turned on and off using the following commands

- **debugprint status** – Prints the status of debug printing on the device.
- **debugprint all_on** – Turns on all debug printing which is compiled into the application.
- **debugprint all_off –** Turns off all debug printing on the device.
- **debugprint on** [cluster id] - Turns on debug printing for the cluster id in question.
- **debugprint off** [cluster id] - Turns off debug printing for the cluster id in question.
- **keys print** - Prints the APS Link Key table. Core printing must be enabled by the define EMBER_AF_PRINT_CORE. If core printing is not enabled, the CLI executes the command but nothing is printed to the command line.
- **print attr** - Prints the ZCL attribute table. Attribute printing must be enabled by the define EMBER_AF_PRINT_ATTRIBUTES. If attribute printing is not enabled, the CLI executes the command but nothing is printed to the command line.
- **print identify** - Prints the identify cluster state. The identify cluster server plugin must be included in the application. If the identify cluster server plugin is not included in the application, the CLI cannot parse the command. Printing for the identify cluster must be enabled by the define EMBER_AF_PRINT_IDENTIFY_CLUSTER. If debug printing is not enabled for the identify cluster server, the CLI executes the command but nothing is displayed on the command line.
- **print groups** - Prints the groups table. The groups cluster server plugin must be included in the application. If the groups cluster server plugin is not included, the CLI cannot parse the command. Also, core debug printing must be enabled by the define EMBER_AF_PRINT_CORE. If core printing is not enabled, the CLI executes the command but nothing is displayed on the command line.
- **print scenes** - Prints the scenes table. The scenes cluster server plugin must be included in the application. If the scenes cluster server plugin is not included, the CLI cannot parse the command. Also, core debug printing must be enabled by the define EMBER_AF_PRINT_CORE. If core printing is not enabled, the CLI executes the command but nothing is displayed on the command line..
- **print c** - The IAS ACE cluster server plugin must be included in the application. If the IAS ACE cluster server plugin is not included, the CLI cannot parse the command. Also, core debug printing must be enabled by the define EMBER_AF_PRINT_CORE. If core printing is not enabled, the CLI executes the command but nothing is displayed on the command line.
- **print d** – Prints the Demand Response Load Control event table. The demand response load control event table must be included in the application, enabled by the define ZCL_USING_LOAD_CONTROL_EVENT_TABLE. If the load control event table is not included, the CLI cannot interpret the command. Also, demand response load control cluster printing must be enabled by the define EMBER_AF_PRINT_DEMAND_RESPONSE_LOAD_CONTROL_CLUSTER. If printing is not enabled for this cluster, the CLI executes the command but nothing is displayed on the command line.

- **print price** - Prints the Price table showing the currently configured prices. The price cluster server plugin must be included in the application. If the price cluster server plugin is not included, the CLI cannot parse the command. Also, price cluster debug printing must be enabled by the define EMBER_AF_PRINT_PRICE_CLUSTER. If price cluster printing is not enabled, the CLI executes the command but nothing is displayed on the command line.
- **print messaging** - Prints the Message table, showing the currently configured or received messages. Either the messaging cluster server or client plugins must be included in the application. If both the messaging cluster server and client plugins are not included, the CLI cannot parse the command. Also, messaging cluster debug printing must be enabled by the define EMBER_AF_PRINT_MESSAGING_CLUSTER. If messaging cluster printing is not enabled, the CLI executes the command but nothing is displayed on the command line.
- **print time** - Prints the current time. The time cluster server must be included in the application, enabled by the define ZCL_USING_TIME_CLUSTER_SERVER. If the time cluster server is not present, the CLI cannot interpret this command.
- **print report** - Prints the reporting table. 'No reporting optimization' must be turned off. If no reporting is defined by ZA_NO_REPORTING_OPTIMIZATION then the CLI cannot interpret this command. Also, report printing must be enabled by the define EMBER_AF_PRINT_REPORTING. If report printing is not enabled, the CLI interprets this command but nothing is displayed on the command line.

11.3.6 Network Commands

- **network form** [channel in decimal or 0x<hex>] [power in decimal or 0x<hex>] [panid 2 byte 0x<hex>] - Starts a ZigBee network. Example: `network form 22 2 0xaabb`
- **network pjoin** [time in seconds in decimal or 0x<hex>] - Enables permit joining on a ZigBee network. Use 0xff to leave joining on permanently. Example: `network pjoin 0xff`
- **network joi**n [channel 1 byte in decimal or <hex>] [power 1 byte in decimal or <hex>] [panid 2 byte 0x<hex>] - Attempts to join to a ZigBee network. The device uses ZA_DEVICE_TYPE when joining. If type is Coordinator it joins as a Router. Example: `network join 22 2 0xaabb`
- **network leave** – Attempts to leave the currently joined ZigBee network. If no ZigBee network is currently joined an error is returned. Example: `network leave`
- **network extpanid** [16 character (8 byte) hex string representing the extended PAN ID desired in hex inside {} without preceding 0x] - Sets the extended PAN ID used for forming or joining a network. This must be done before a device is part of a network. Example: `network extpanid { aa bb cc dd aa bb cc dd }`
- **option disc** [clusterId as 4 character (2 byte) unsigned integer in decimal or 0x<hex>] - Sends a ZDO Match Descriptor Request for the server side of the cluster specified. Match Descriptor Responses received are printed to the serial output.
- **option edb** [endpoint as 1 byte decimal or 0x<hex>]- Sends a ZDO End Device Bind Request using the local endpoint specified by ZA_GLOBAL_SRC_ENDPOINT.

11.3.7 Security Commands

- **keys print** - Prints the APS Link Key table.
- **keys clear** - Clears the APS Link Key table.
- **option link** [index in key table decimal or 0x<hex>] [EUI64 in little endian format in hex inside {} without preceding 0x] [key bytes 0-15 in hex inside {} without preceding 0x] - Sets a link key in the link key table. Example: `option link 0x00 { 06 00 ab 41 64 30 00 0a } {aa bb cc dd ee ff aa bb cc dd ee ff aa bb cc dd}`
- **option registe**r - Initiates Smart Energy Registration including Key Establishment. This command expects that the device has already joined a smart energy network, but is not yet registered.
- **option partner** [partner EUI in hex inside {} without preceding 0x] - Initiates a partner link key request to the Trust Center (available only when using Smart Energy Security).

11.3.8 Commands for Building and Sending Messages

- **send** [id as 4 character hex string 0x<hex>] [src endpoint in decimal or 0x<hex>] [dst endpoint in decimal or 0x<hex>] - Sends a message using the message that exists in the current send buffer (created using "zcl" command).
- **bsend** [src endpoint in decimal or 0x<hex>] - Sends the current message to any device in the binding table that has a binding entry with a cluster matching the cluster specified in the current message, and a local endpoint matching the

source endpoint provided as the argument to this call. If the source endpoint is specified as 0, this is interpreted as a special value indicating that the current message should be sent to any and all bindings found in the binding table that match the built message's cluster ID.

- **timesync** [id as 4 character hex string or 0x<hex>] [src endpoint in decimal or 0x<hex>] [destination endpoint in decimal or 0x<hex>] - Syncs time with the device specified by sending a read attribute command and using the read attribute response.
- **anon short** [id as 4 character 0x<hex>] [short dest panid as 4 character 0x<hex>] [dest app-profile-id as 4 character 0x<hex>] - Sends an Inter-PAN message using the message that exists in the send buffer (created with "zcl" command) to the shortID specified on the destination PAN specified and using the Application Profile specified.
- **raw** [clusterID in decimal or 0x<hex>] [All data in string or hex array inside curly braces {}] - Creates a message by specifying the raw bytes. To specify messages larger than 32 bytes use this command first and then use "raw2". Use "send" to send the message once it has been created. Example: `raw 0x0f {00 0A 00 11 22 33 44 55}` sends a message to cluster 15 (0xF) of length 8, which includes the ZCL header.

11.3.9 ZCL Global Serial Commands

These messages only construct the message payload. The message must still be sent using the "send" command. The "zcl global direction" command can be used to change the direction (client-to-server or server-to-client) of future ZCL global commands. By default, commands are sent client-to-server.

- **zcl global read** [cluster in decimal or 0x<hex>] [attrID in decimal or 0x<hex>]
- **zcl global write** [cluster in decimal or 0x<hex>] [attrID in decimal or 0x<hex>] [data type in decimal or 0x<hex>] [data (variable size depends on data type) as a "<string>" or {<hex array>}]
- **zcl global uwrite** [cluster in decimal or 0x<hex>] [attrID as decimal or 0x<hex>] [data type as decimal or 0x<hex>] [data as a "<string>" or {<hex array>}]
- **zcl global nwrite** [cluster in decimal or 0x<hex>] [attrID as decimal or 0x<hex>] [data type as decimal or 0x<hex>] [data as a "<string>" or {<hex array>}]
- **zcl global discover** [cluster in decimal or 0x<hex>] [attrID as decimal or 0x<hex>] [max # to report in decimal or 0x<hex>]
- **zcl global report-rea**d [cluster in decimal or 0x<hex>] [attrID as decimal or 0x<hex>] [direction in decimal or 0x<hex> (0 for client-to-server, 1 for server to client)]
- **zcl global send-me-a-report** [cluster in decimal or 0x<hex>] [attrID in decimal or 0x<hex>] [data type as decimal or 0x<hex>] [min report time as decimal or 0x<hex>] [max report time as decimal or 0x<hex>] [reportable change as a "<string>" or {<hex array>}]
- **zcl global expect-report-from-me** [cluster in decimal or 0x<hex>] [attrID as decimal or 0x<hex>] [timeout in decimal or 0x<hex>]
- **zcl global direction** [direction in decimal or0x<hex> (0 for client-to-server, 1 for server to client)]

11.4 ZDO Commands

- **zdo simple** [2 byte dest] [1 byte target ep] - Sends a simple description request to the chosen destination and endpoint.
- **zdo node** [2 byte dest] - Sends a node descriptor request to the destination specified.
- **zdo match** [2 byte dest] [2 byte profile] - Sends a match descriptor request to the destination specified.
- **zdo bind** [2 byte dest] [1 byte local ep] [1 byte remote ep] [2 byte cluster] [8 byte THEIR eui] [8 byte dest eui] - Sends a bind request to the destination specified for the given arguments.
- **zdo in-cl-list add** [array of two byte cluster IDs] - Adds to the "in" or "server" cluster list.
- **zdo in-cl-list clear** - Clears the "in" or "server" cluster list.
- **zdo out-cl-list add** [array of two byte clusters IDs] - Adds to the "out" or "client" cluster list.

- **zdo out-cl-list clear** - Clears the "out" or "client" cluster list.
- **zdo nwk-upd chan** [1 byte channel] - Sends an update channel request.
- **zdo nwk-upd set** [2 byte nwk mgr ID] [4 byte chan mask] - Sends a set network manager request for the given network manager node ID and active channels provided in the 4-byte channel mask.
- **zdo nwk-upd scan** [2 byte target node ID] [1 byte scan duration] [2 byte scan count] - Performs an energy scan.
- **zdo active** [2 byte target node ID] - Sends an active endpoint request to the short address provided.

11.5 ZCL Cluster-Specific Commands

The following commands are implemented on a per-cluster basis and can be used to perform basic tasks for that cluster. The cluster in question must be included in the application for these commands to be included. If the cluster in question is not included, the CLI cannot interpret the command.

11.5.1 ZCL Basic Client Commands

- **zcl basic rtfd - Builds a Reset to Factor Defaults command.**

11.5.2 ZCL Identify Client Commands

- **zcl identify id** [identify time as 4 character hex string]
- **zcl identify query**

11.5.3 ZCL Identify Server Commands

- **zcl identify on** [seconds in decimal] - Turns on identify on the local device for the number of seconds specified.
- **zcl identify off** - Turns off identify on the local device.

11.5.4 ZCL Groups Client Commands

- **zcl groups add [**groupId as 4 character hex string] [name as 16 character ascii string]
- **zcl groups view** [groupId as 4 character hex string]
- **zcl groups get** [count in decimal] [[groupId as 4 character hex string] * count]
- **zcl groups remove** [groupId as 4 character hex string]
- **zcl groups rmall -** Creates a ‘remove all groups’ command.
- **zcl groups ad-if-id** [groupId as 4 character hex string] [name as 16 character ascii string] - Creates an Add If Identify command.

11.5.5 ZCL Scenes Client Commands

- **zcl scenes add** [groupId as 4 character hex string] [sceneId as 2 character hex string] [transition time as 4 character hex string] [name as 16 character ascii string]
- **zcl scenes view** [groupId as 4 character hex string] [sceneId as 2 character hex string]
- **zcl scenes remove** [groupId as 4 character hex string] [sceneId as 2 character hex string]
- **zcl scenes rmall** [groupId as 4 character hex string]
- **zcl scenes store** [groupId as 4 character hex string] [sceneId as 2 character hex string]
- **zcl scenes recall** [groupId as 4 character hex string] [sceneId as 2 character hex string]
- **zcl scenes get-membership** [groupId as 4 character hex string]
- **zcl scenes set** ["on" or "off" keyword] [level value as integer] - Sets the scene extensions for on/off and level control.

11.5.6 ZCL On/Off Client Commands

- **zcl on-off off**
- **zcl on-off on**
- **zcl on-off toggle**

11.5.7 ZCL Level Control Client Commands

All arguments are given in hex. The value listed after an argument is the number of bytes it expects. The number of characters expected is 2 times that number.

- **zcl level-control mv-to-level** [level:1] [transition time:2]
- **zcl level-control move** [mode:1] [rate:1]
- **zcl level-control step** [step:1] [step size:1] [trans time:2]
- **zcl level-control stop**
- **zcl level-control o-mv-to-level** [level:1] [trans time:2]
- **zcl level-control o-move** [mode:1] [rate:1]
- **zcl level-control o-step** [step:1] [step size:1] [trans time:2]
- **zcl level-control o-stop**

11.5.8 ZCL Time Server Commands

- **zcl time** [year] [month] [day] [hour] [min] [sec] - Sets the local time to the value specified; all arguments in decimal.

11.5.9 ZCL Thermostat Client Commands

- **zcl tstat set** [mode as 2 character hex string] [amount as 2 character hex string]

11.5.10 ZCL IAS Zone Server Commands

- **zcl ias-zone enroll** [zone type as 4 character hex string] [manuf code as 4 character hex string]
- **zcl ias-zone sc** [zone status as 4 character hex string] [extended status as 2 character hex string]

11.5.11 ZCL IAS ACE Client Commands

- **zcl ias-ace arm** [mode as 2 character hex string]
- **zcl ias-ace bypass** [numZones as 2 character hex string] [zones: numZones zones as 2 character hex string]
- **zcl ias-ace emergency**
- **zcl ias-ace fire**
- **zcl ias-ace panic**
- **zcl ias-ace getzm**
- **zcl ias-ace getzi** [zoneID as 2 character hex string]

11.5.12 ZCL Price Client Commands

- **zcl price current**
- **zcl price scheduled** [startTime as 8-character (4-byte) hex string representing timestamp in seconds] [number of events requested, as 1-byte decimal value, in the range 0-5; 0 means send all scheduled prices]

11.5.13 ZCL Price Server Commands

These commands are used for configuring prices in the Price table on the ESP. The Price table can be viewed by using "print price"

- **zcl set-price clear**

- **zcl set-price who** [index as decimal] [providerID as 8 character (4 byte) hex string] [label as up to 13 character ascii string] [eventId as 8 character (4 byte) hex string]
- **zcl set-price what** [index as decimal] [unitOfMeas as 2 character (1 byte) hex string] [currency as 4 character (2 byte) hex string] [ptd as 2 character (1 byte) hex string] [PTRT as 2 character (1 byte) hex string]
- **zcl set-price when** [index as decimal] [startTime as 8 character (4 byte) hex string] [duration as 4 character (2 byte) hex string]
- **zcl set-price price** [index as decimal] [price as 8 character (4 byte) hex string] [ratio as 2 character (1 byte) hex string] [genPrice as 8 character (4 byte) hex string] [genRatio as 2 character (1 byte) hex string]
- **zcl sprice publish** - Sets up an unsolicited publish price command.

11.5.14 ZCL Demand Response Load Control Server Commands

- **zcl drlc lce** [eventId as 8 character (4 byte) hex string] [start time as 8 character (4 byte) hex string] [duration in minutes as 4 character (2 byte) hex string] [event control as 2 character (1 byte) hex string] - Constructs a load control event.
- **zcl drlc slce** [index (1 byte) decimal or hex] [length (1 byte) decimal or hex] [load control event bytes] – Schedules a load control event. The Index represents the index in the DRLC Server plugin's scheduled load control event table. The length indicates the number of bytes in the load control event and is expected to be 23. Load control event bytes are expected as 23 raw bytes in the form {<eventId:4> <deviceClass:2> <ueg:1> <startTime:4> <duration:2> <criticalityLevel:1> <coolingTempOffset:1> <heatingTempOffset:1> <coolingTempSetPoint:2> <heatingTempSetPoint:2> <afgLoadPercentage:1> <dutyCycle:1> <eventControl:1> }. All multi-byte values should be little endian as though they were coming over the air. Example:

  ```
  zcl drlc slce 0 23 {ab 00 00 00 ff 0f 00 00 00 00 00 01 00 01 00 00 09 1a 09 1a 0a 00}
  ```

- **zcl drlc sslce** [index (1 byte)] – Sends a scheduled load control event. Loads a load control event out of the DRLC server's scheduled load control event table into the send buffer so that it can be sent over the air. The user must still use the send command to actually send the load control event over the air.
- **zcl drlc cslce –** Clears all scheduled load control events from the server's load control event table.
- **zcl drlc cl** [eventId as 8 character (4 byte) hex string] [device class 2 bytes hex or decimal] [utility enrollment group 1 byte hex or decimal] [cancel control 1 byte hex or decimal] [start time as 8 character (4 byte) hex string] - Cancels a load control event.
- **zcl drlc ca** - Cancels all load control events.

11.5.15 ZCL Demand Response Load Control Client Commands

- **zcl drlc gse** [number of events to get as 2 character (1 byte) hex string] - Get Scheduled Events command.
- **zcl drlc opt in** [eventId 4 bytes] - Opts in to the DRLC event indicated by eventId.
- **zcl drlc opt out** [eventId 4 bytes] - Opts out of the DRLC event indicated by eventId.

11.5.16 ZCL Simple Metering Client Commands

- **zcl sm gp** [type as 2 character (1 byte) hex string] [time as 8 character (4 byte) hex string] [intervals as 2 character (1 byte) hex string] - Get Profile.

11.5.17 ZCL Simple Metering Server Commands

These commands are used on an ESP or a Meter to simulate the behavior of a meter.

- **zcl tm print** - Prints the state of the Test Meter setups
- **zcl tm** [electric | gas | off] - Sets the Test Meter mode. electric or gas turn the test mode on. off turns the test mode off.
- **zcl tm rate** [consumption rate in decimal] - Sets the normal consumption rate per second.
- **zcl tm variance** [variance in decimal] - Sets a variance to add to the normal consumption rate each second. If the consumption rate is set to 2 and variance is set to 3, the consumption rate will be between 2 and 5 units per second. The actual number is randomly chosen.

- **zcl tm adjust** - Adjusts the consumption of the meter based on the time value as if the device was consuming at the currently set rate and variance for the whole of the current day .

11.5.18 ZCL Messaging Client Commands

- **zcl message get** - sends a get last message command.
- **zcl message confirm** - Sends a message confirmation command.

11.5.19 ZCL Messaging Server Commands

- **zcl message display** - Sends a display message using the current message.
- **zcl message cancel** - Sends a cancel message using the current message.

The following commands are for setting the message on the Messaging Server

- **zcl set-message message** [message up to 16 ascii character string]
- **zcl set-message append** [message up to 16 ascii character to append to the current message]
- **zcl set-message id** [messageId as 8 character (4 byte) hex string]
- **zcl set-message time** [start time as 8 character (4 byte) hex string] [duration as 4 character (2 byte) hex string]
- **zcl set-message transmission** [normal|ipan|both]
- **zcl set-message importance** [low|med|high|critical]
- **zcl set-message confirm** [true|false]
- **zcl set-message cancel**

11.5.20 ZCL OTA Client Commands

- **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 start -** Starts the OTA client state machine. The state machine discovers the OTA server, queries for new images, downloads the images, and waits for the server command to upgrade.
- **zcl ota client stop -** Stops the OTA client state machine.

11.5.21 ZCL OTA Server Commands

- **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 polices used by the OTA Server Policy Plugin.
- **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> -** Sends an OTA Image Notify message to the specified destination indicating that 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

12 The Debug Printing Interface

Application Framework V2 includes a new, more granular, debug printing interface. As a result, 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 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 Porting Applications

13.1 Porting an Application from AFV2 Beta I to AFV2 Beta II, Beta III and the GA release:

To port applications created under Application Framework V2 (AFV2) Beta I or Beta II you must create a new configuration ".isc" file in AppBuilder, regenerate your application, and copy your callback code over to the new generated callback signatures.

Changes between AFV2 Beta II and AFV2 Beta III

1. **Plugins.** All of the code which used to reside under app/framework/cluster (with the exception of Time and Key Establishment) has been moved into the app/framework/plugins directory. Now you can optionally include or exclude Ember-created cluster implementations. If you include a plugin for a specific cluster, that plugin will consume the command callbacks from within the framework that it cares about. When the plugin is selected, each callback that it uses is marked within the AppBuilder interface as being implemented by a plugin. AppBuilder does not allow you to generate a configuration that includes command callbacks implemented by both a plugin and the callbacks stub file, since this would create a linker error.
2. **Event management:** All of the plugin code has been moved to an event system for application events. This code used to rely on a tick mechanism for running periodic application services. The tick mechanism did not work well for sleepy devices, which need to know reliably when they need to wake up to service some application function. The new event mechanism allows the framework to keep track of all the cluster events so that it knows when it can go to sleep and for how long without disrupting the function of the application.
3. **Default response transmission:** See section 10.3, Sending a Default Response.
4. **Removed emberAfPrint with newline argument.** This function has been replaced with two functions, emberAfPrint() and emberAfPrintln(). This provided a significant flash savings.
5. **preMessageReceivedCallback updated:** This callback now provides a pointer to the incoming message as an EmberAfIncomingMessage struct. This structure contains all of the incoming message data and allows the Application Framework to provide a single callback interface for both Host and SoC devices.
6. **EmberAfStatus used instead of int8u:** Some of the callbacks which receive status bytes have been changed to receive an EmberAfStatus parameter instead of an int8u. This makes it possible for us to change the status byte in the future if we wish to.
7. **Removed the custom emberAfDemandResponseLoadControlCluster-DRLCGetScheduledEventsCallback**: This custom callback has been replaced with the generated callback of the same name.
8. **Added write permission enum used by emberAfAllowNetworkWriteAttribute.** This enum is listed in app/framework/gen/ and is used to indicate one of three possible values returned from the emberAfAllowNetworkWriteAttribute callback: DENY_WRITE, ALLOW_WRITE_NORMAL, and ALLOW_WRITE_READONLY. For more information on the emberAfAllowNetworkWriteAttribute callback and its use see section 6.12.
9. **Removed emberAfClusterTickCallback:** All of the cluster code has been moved into plugins and modified to use the Ember Event mechanism for scheduling periodic events. Once this was done, emberAfClusterTickCallback was not needed, since the cluster ticks are no longer called through the tick mechanism. All code which was implemented in the emberAfClusterTickCallback should be moved over to emberAfMainTickCallback, which is called on every tick.

   Any code which previously relied on the timing of the cluster tick callback consistent with the timing of the callback in the plugin should now keep track of its own time by using the various HAL functions to retrieve the current time and check how much time has passed. In general, however, all application code should use the Ember Event mechanism in place of relying on being called on every tick. This is especially true if your device is a sleepy end device, since it will not be able to rely on the application calling its tick on an ongoing basis (see Chapter 14 for a

more detailed discussion of sleepy end devices). The event mechanism allows you to indicate how long the device may nap for and to rely on it being called when the event is scheduled.

Changes between AFV2 Beta I and AFV2 Beta II

The following lists of some of the updates in AFV2 Beta II and how they affect porting an application from AFV2 Beta I to Beta II.

1. **Attribute storage options.** The addition of attribute storage options such as external attributes and singletons means that existing AFV2 endpoint configuration files will not work with AFV2 Beta II. They need to be regenerated from AppBuilder.
2. **A more consistent API.** Many of the interfaces that AFV2 Beta I customers used still exist in the AFV2 API, however their signatures have changed to be more consistent with the naming strategy for prefixes in the EmberZNet API. Thus functions, types, and some globally accessible variables now start with the 'emberAf' prefix. The publicly available APIs have also been moved into a central location (app/framework/include/af.h) to be consistent with the way that the EmberZNet API is bundled.
3. **More callbacks.** AFV2 Beta II contains new callbacks. For the most part these are cluster-specific callbacks related to command handling. Every cluster command now includes a command callback. Some commands are still handled internally by bundled Ember cluster code. For example, DRLC and Key Establishment commands are handled by the code located in app/framework/cluster/demand-response-load-control.c and app/framework/cluster/key-establishment.c, respectively. All commands that include a default Ember implementation in the cluster code are marked with a (+) symbol in AppBuilder's callback tab. For these commands, if the callback is implemented it must return a value of FALSE for the internal code to continue to handle the command. A return value of TRUE indicates that the command has been completely handled and that the internal Ember code should not continue to process the command.
4. **Callback signature changes**. Callback signatures have been updated to reflect the consistent API emberAf prefix. For example, the preAttributeChangeCallback still takes the same arguments but the name has changed to emberAfPreAttributeChangeCallback
5. **readAttributeValueFromHWCallback and writeAttributeValueToHWCallback have been deleted.** These callbacks were no longer necessary after the ability to 'externalize' attributes was added. Any attribute values that are stored in hardware should be marked as external so that they can be accessed through the two associated callbacks emberAfExternalAttributeReadCallback and emberAfExternalAttributeWriteCallback. Once an attribute is marked as external in AppBuilder, these two callbacks are included automatically, since they are then required for the proper operation of attribute storage.
6. **preCommandReceivedCallback changed to emberAfPreCommandReceivedCallback and now passes EmberAfClusterCommand struct.** The related command information is parsed out into a struct for easy access. The same struct that is passed to emberAfPreCommandReceived is available at any time during command processing through the global utility macro emberAfCurrentCommand() defined in app/framework/include/af.h
7. **Less cluster code.** Any code in app/framework/cluster that had previously been included as a simple place holder for command handling or attribute management has been removed. Many of the files in app/framework/cluster contained command handling code that simply printed out the incoming command data. All of these handlers have now been removed in favor of cluster-specific callbacks called from app/framework/gen/call-command-handler.c. Furthermore, a command will only be handled if it either includes a bundled Ember handler or a callback. If neither is implemented, the command is considered unsupported and will return a default response of 'unsupported command'.

13.2 Porting an Application from AFV1 to AFV2

The overall design concept of the Application Framework remains unchanged from AFV1 (located in app/ha) to AFV2. You still design your application using AppBuilder's GUI interface and then generate configuration and build files. The process of porting is as follows:

1. Create your new application in AppBuilder and generate the configuration files as you did with V1.
2. Any code that you wrote for your V1 application needs to be copied out of the V1 framework files and placed in the appropriate generated callback. You may need to revisit your application configuration and regenerate your configuration files a few times before you figure out all the callbacks you wish to implement.
3. The following is an example from a port from V1 application to AFV2. It lists the callbacks implemented in the AFV2 application. Each callback lists the code that was placed in that callback and where it previously resided in V1.

- emberAfPreMessageReceivedCallback: The application had some ZDO and service discovery code written directly into the emberIncomingMessageHandler in the main file. This was copied and placed inside the emberAfPreMessageReceived callback.
- emberAfCommandReceivedCallback: Some code was implemented directly into the command processing handlers in zcl-util.c. This was copied and moved into the emberAfCommandReceivedCallback.
- emberAfMainTickCallback: Some code needed to be periodically checked, and was located directly in the V1 main loop. This was copied into the emberAfMainTickCallback.
- emberAfClusterInitCallback: Some of the attributes that are stored in RAM need to be initialized to a specific value based on the type of device. These initializations used to be done in the c file for the cluster that needed initialization. This initialization code was copied into the emberAfClusterInitCallback.
- emberAfMainInitCallback: Some code to set up the application configuration was written directly into the main function. This was copied out of the actual main and into the emberAfMainInitCallback.
- Message buffer loading: The application built message buffers for sending commands. This logic was replaced by the new command macros for message buffer loading. All of the associated command macros are located in app/framework/gen/client-command-macro.h.

14 Sleepy Devices

14.1 Introduction

The 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.

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 and the End Device Poll timeout. These two timeouts correspond to two polling intervals on the sleepy end device: the SHORT_POLL and the LONG_POLL intervals; previously called the “nap duration” and “hibernation duration” respectively. We have chosen to move away from the terms nap and hibernate since they are not descriptive of the purpose for the interval values themselves.

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 ZigBee APS retry timeout. 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 AppBuilder interface has no GUI widget to allow users to manipulate this value. It defaults to one second. However, the user can define this value in the macros section of the includes tab within the AppBuilder interface.

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.

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 based 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: There is no standard way to timeout entries in a child table in the ZigBee protocol. 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. Ember has developed a more deterministic mechanism for child aging called the "End Device Poll Timeout." Ember parents expect 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, Ember has a configured an assumed upon 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 which 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 which 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.

14.2.3 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 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 Application Framework automatically checks with the event mechanism to see when the next application event is scheduled. The 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 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 looses 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 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 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 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/

15 Application Framework Plugins

15.1 Introduction

The 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 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 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
    a. Disable the plugin and implement the callbacks you need for your application.
    b. This will add the callbacks into your application_callbacks.c file.
2. Create your own plugin from the original
    a. Go to the plugin directory within the stack install located at app/framework/plugin.
    b. Copy the plugin contents into a new directory inside app/framework/plugin.
    c. Each plugin includes a configuration file called plugin.properties. This file is used by 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 AppBuilder.
    d. Once you do this, you must restart AppBuilder so that your new plugin is picked up when 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 Application Framework V2.

15.3.1 Architecture

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

Ember 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.

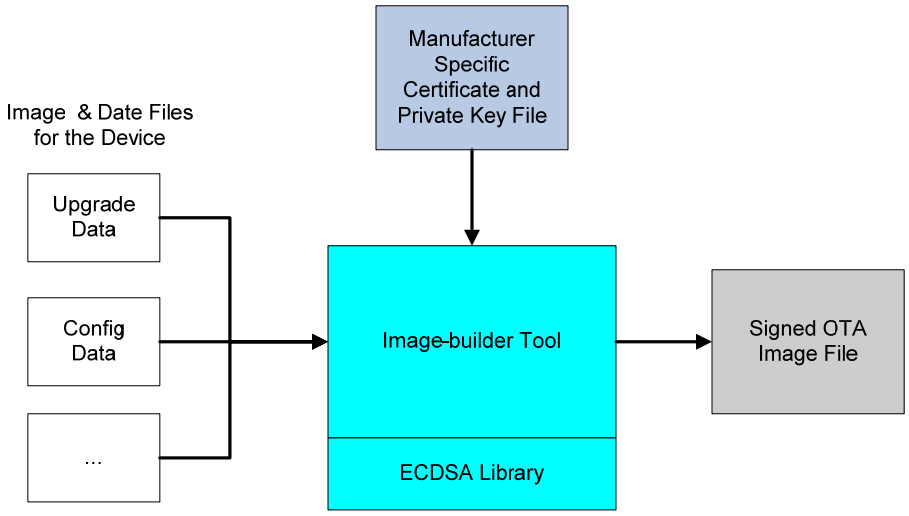
1. What external storage device will be used for the OTA upgrade image?
    a. Ember provides a few EEPROM driver implementations as well as a POSIX file system (for UART host only).
    b. If a different driver or method is desired, then this code must be provided.
2. Does a client device require multiple upgrade files in order to bootload?
    a. 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.
    b. The ZigBee OTA cluster also supports requesting multiple files, but the client must manage this.
3. How will upgrade files be labeled?
    a. 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?
    a. 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.
    b. 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?
    a. Bootloading is device specific, though Ember 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?
    a. 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

Ember 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 15-1. 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 the Image-builder documentation in the stack release directory: **tool/image-builder/Image-Builder-Instructions.pdf.**

Figure 15-1. 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 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:

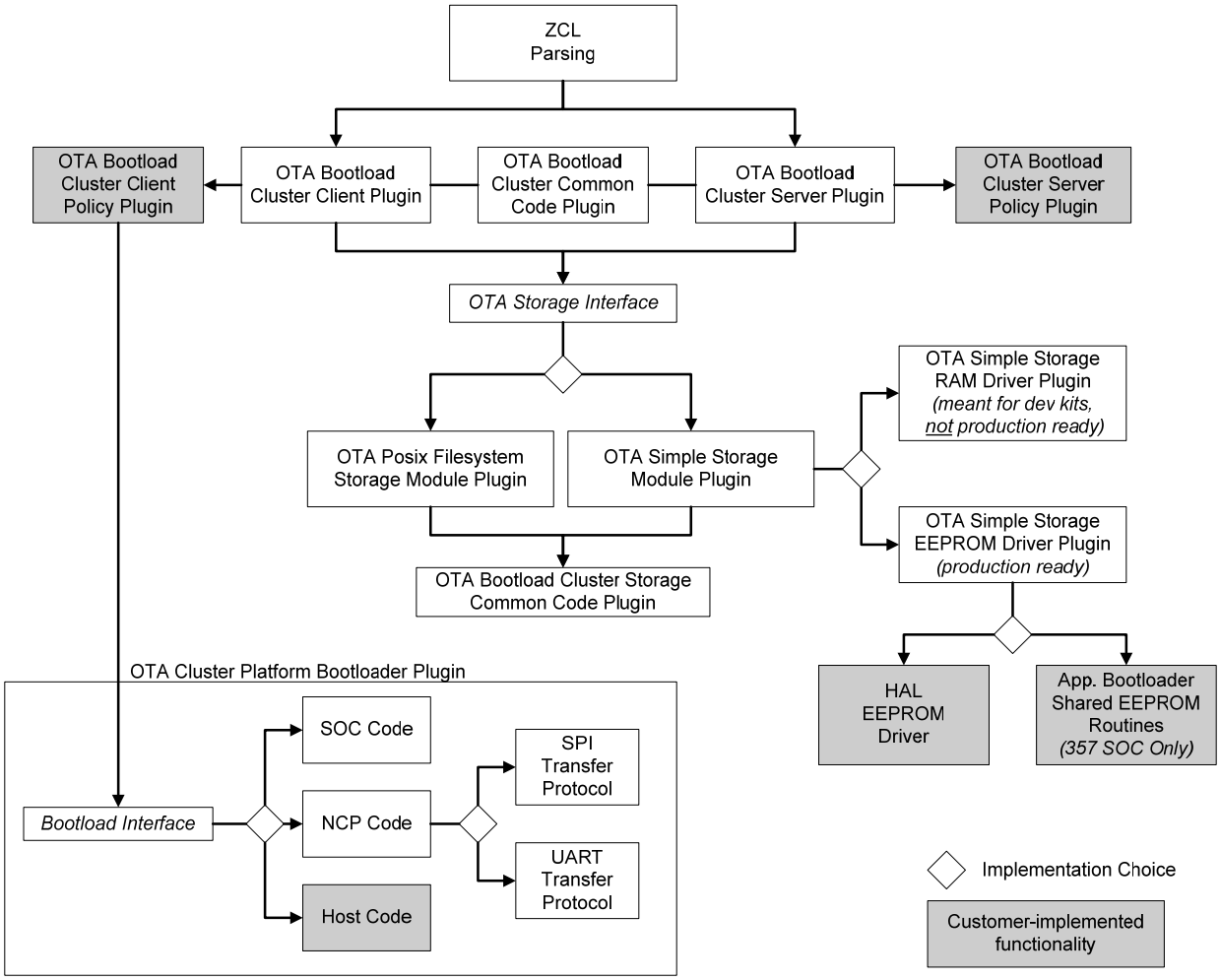
1. Certificates and private keys used for signing should only be used for signing. They should never be put on devices as device-specific certificates and keys. This holds true regardless of whether the device is a test device or a production device.
2. The EUI used for the signing certificate should 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 15-2.

Figure 15-2. 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.

Ember provides an 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.

Ember provides an Application Framework plugin that implements the client cluster. The Ember 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 Ember. 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.

Ember 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 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.

Ember 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**.

Ember's 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.

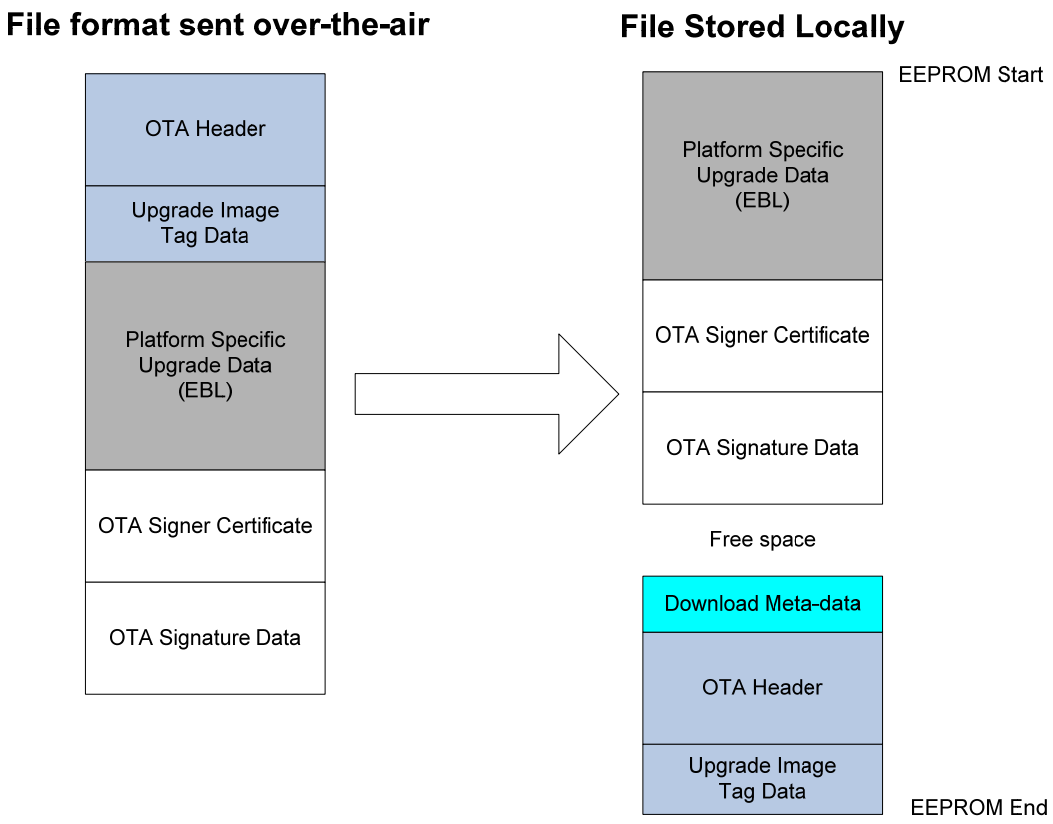
15.3.9.4 OTA Simple Storage EEPROM Driver Plugin

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 15-3 illustrates a change in the OTA image on disk.

Figure 15-3. 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 4.2 application bootloader must enable this compatibility mode and include a copy of a newer EEPROM driver (EmberZNet 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 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 EERPOM.
OTA Storage End Offset	The last offset for the OTA image storage location in the EEPROM.

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.

Ember 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, Ember 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 Configuring the OTA Cluster

15.3.11.1 Configuring an OTA Client

Follow these steps to configure an over the air bootload cluster client application. These steps assume that the OTA storage for the client is either an external EEPROM or a POSIX filesystem.

1. From the File menu, Select New → Zigbee Application Configuration.
2. An Install Stack Selection pop-up window is displayed. Select an EmberZNet 4.3.0 GA, or later, stack according to your hardware platform.
3. On the ZCL Cluster Configuration tab, change the ZCL Device Type according to the device you wish to create.
4. Under the General category of clusters, select the Over the Air Bootloading cluster client.

   Note: Certain attributes that are optional in the ZigBee OTA specification are required by the Ember client plugin; they are automatically turned on.
5. Optional: Turn on OTA debug printing
   a. Go to the **Stack** configuration tab.
   b. In the **Debug Printing** section, expand the **Cluster Debugging** group.
   c. Select **Compiled in** and **Enabled at Startup** for the **Over the Air Bootloading Cluster**.
6. If the device is an EM250 or EM35x SOC device, add a bootloader.
   a. On the HAL configuration tab, select the **Application** bootloader.

      Note: Only an application bootloader is supported by the OTA cluster.
7. Go to the **Plugins** configuration tab and enable the following plugins:
   a. OTA Cluster Platform Bootloader

b. OTA Bootload Cluster Client

c. OTA Bootload Cluster Client Policy

d. OTA Bootload Cluster Common Code

e. OTA Bootload Cluster Storage Common Code

8. If the configuration is for an NCP UART then do the following:

   a. Enable the OTA POSIX Filesystem Storage Module plugin.

9. If the configuration is anything else, do the following:

   a. Enable the OTA Simple Storage Module plugin.

   b. Enable the OTA Simple Storage EEPROM Driver plugin.

   c. Go to the Includes configuration tab.

      i. Under the **Code Files** section, add in the driver file that implements the HAL EEPROM functions.

      ii. If your device is an EM357 or EM351 with an **EmberZNet 4.3.0 GA**, or later, application bootloader, skip the above step and enable use of the shared EEPROM download routines. See below.

   a. If the device is an SOC application do the following.

      iii. Under the OTA Simple Storage EEPROM Driver plugin, select the option SOC Bootloading Support.

      iv. If your device is an EM35x with an EmberZNet 4.2.0 GA application bootloader, select the option **Enable 4.2 Application Bootloader Compatibility Mode**.

10. Perform any additional application configuration.

11. Click the **Generate** button.

15.3.11.2 Configuring an OTA Server

Follow these steps to configure an over the air bootload cluster server application. These steps assume that the OTA storage for the client is either an external EEPROM or a POSIX file system.

1. From the File menu, select New → ZigBee Application Configuration.

2. An Install Stack Selection pop-up window is displayed. Select an EmberZNet 4.3.0 GA, or later, stack according to your hardware platform.

3. On the ZCL Cluster Configuration tab, change the ZCL Device Type according to the device you wish to create.

4. Under the General category of clusters, select the Over the Air Bootloading cluster server.

5. Optional: Turn on OTA debug printing.

   a. Go to the **Stack** configuration tab.

   b. In the **Debug Printing** section, expand the **Cluster Debugging** group.

   c. Select **Compiled in** and **Enabled at Startup** for the **Over the Air Bootloading Cluster**.

6. Go to the **Plugins** configuration tab and enable the following plugins:

   a. OTA Bootload Cluster Server

   b. OTA Bootload Cluster Server Policy

   c. OTA Bootload Cluster Common Code

   d. OTA Bootload Cluster Storage Common Code

7. If the configuration is for an NCP UART then do the following:
    a. Enable the OTA POSIX Filesystem Storage Module plugin.
8. If the configuration is anything else, do the following:
    a. Enable the OTA Simple Storage Module plugin.
    b. Enable the OTA Simple Storage EEPROM Driver plugin.
    c. Depending on your platform, you must now include the EEPROM driver and selection options within the OTA Simple Storage EEPROM Driver plugin according to the following chart:

	SPI Host	SOC Em250	SOC Em35x w/ EmberZNet 4.2.0 GA	SOC Em35x w/ EmberZNet 4.3.0 or later.
Include EEPROM Driver[1]	X	X	X	
Enable SOC Bootload Support[2]		X	X	X
Compatibility Mode[3]			X	

[1]To include the EEPROM driver for your platform do the following: Go to the **Includes** configuration tab. Under the **Code Files** section, add in the driver file that implements the HAL EEPROM functions.

[2]To enable SOC Bootload support do the following: Go to the Plugins tab. In the **OTA Simple Storage EEPROM Driver**, Select the option **SOC Bootloading Support**.

[3]To enable EmberZNet 4.2.0 compatibility mode do the following: Go to the Plugins tab. In the **OTA Simple Storage EEPROM Driver** Select the option **Enable 4.2 Application Bootloader Compatibility Mode**.

9. Perform any additional application configuration.
10. Click the **Generate** button.

15.3.12 OTA Cluster Command Line Interface

15.3.12.1 Client Commands

Table 15-1. 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.
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 AppBuilder. By default, if the client enabled page request support in 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.

15.3.12.2 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.

15.4 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 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.

InSight 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 (Ember's 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 Ember's 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 Ember's manufacturer-specific attribute.

Note: Ember's 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

16.1 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 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 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 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.

16.2 Defining ZCL Extensions within the Application Framework and 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 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 AppBuilder by following the instructions included in the *Ember AppBuilder User's Guide* (Document 120-4035-000) and the 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 Application Framework and AppBuilder located at tool/appbuilder/appbuilder.xsd. Further documentation about extending the 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.

16.3 Manufacturer-Specific Attribute APIs

Some APIs in Application Framework used to interact with attributes have been modified to take a manufacturer code as an argument.

16.3.1 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.

16.3.2 Attribute Changed Callbacks

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

16.3.3 Sample Application

The SeMeterMirror sample application uses an Ember 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 Ember's manufacturer code 0x1002 we were able to leverage the 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.