UG103.5

EMBER® APPLICATION DEVELOPMENT FUNDAMENTALS: SECURITY

This document introduces some basic security concepts, including network layer security, trust centers, and application support layer security features. It then discusses the types of standard security protocols available in EmberZNet PRO. Coding requirements for implementing security are reviewed in summary. Finally, information on implementing ZigBee Smart Energy security is provided.

New in This Revision

Initial release (contents previously published).

Contents

1 Introduction

Security is a major concern in the ZigBee architecture. Although ZigBee uses the basic security elements in IEEE 802.15.4 (for example, AES encryption and CCM security modes), it expands upon this with:

- 128-bit AES encryption algorithms
- Strong, NIST-approved security
- Defined key types (link, network)
- Defined key setup and maintenance
- Keys can be hardwired into an application
- CCM* (Unified/simpler mode of operation)
- Trust centers
- Security that can be customized for the application

As Figure 1 illustrates, the security services provider block interactions with both the application and network layers.

Two levels of security have been defined in the ZigBee PRO specification: standard and high. Standard security is a superset of the ZigBee 2006 residential security, and is intended to be fully backward-compatible with 2006 devices operating as end devices. High security is not currently used by any existing ZigBee application profiles and is not supported in any release of the EmberZNet stack library, so discussion of that model is beyond the scope of this document. Also note that IEEE 802.15.4 MAC-level security is not used by ZigBee and is therefore not supported by EmberZNet PRO and not described here.

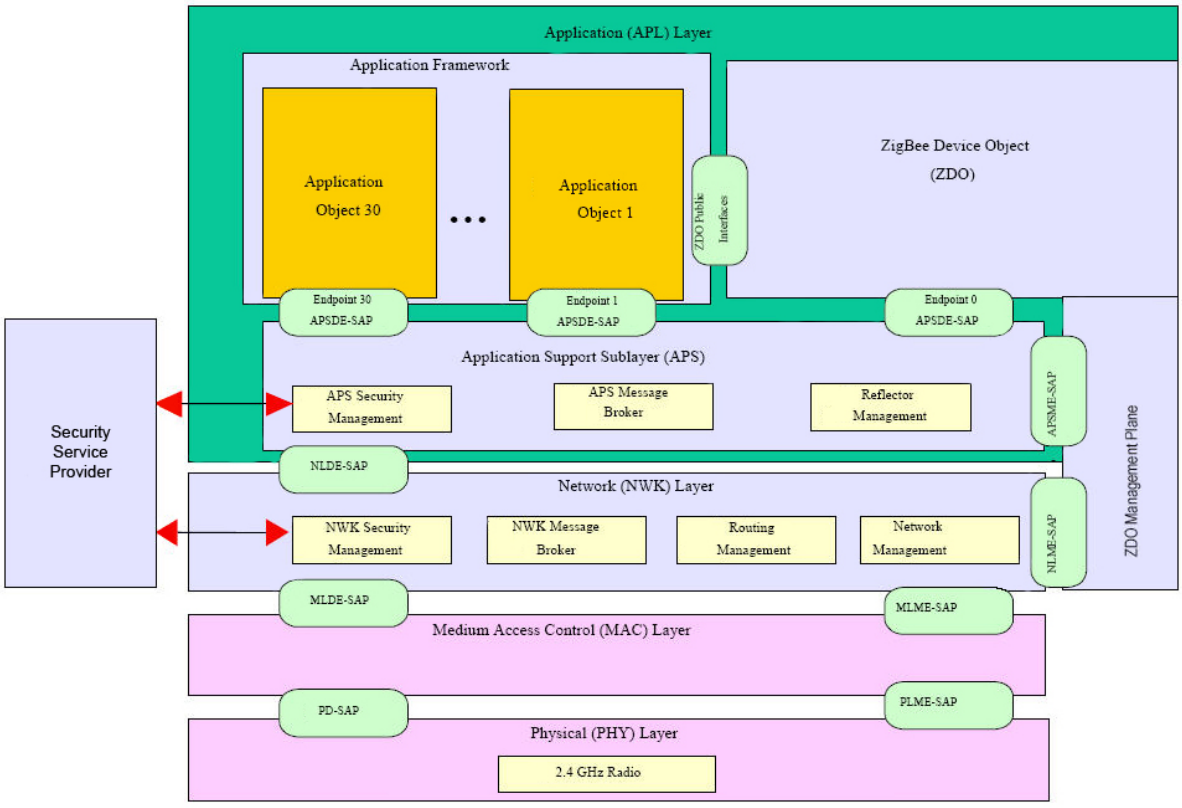

Figure 1. ZigBee Stack Architecture

This document first describes some basic security concepts, including network layer security, trust centers, and application support layer security features. It then discusses the types of standard security protocols available in EmberZNet PRO. Coding requirements for implementing security are reviewed in summary. Finally, information on

implementing ZigBee Smart Energy security is provided. Details may be found in document AN714, *Smart Energy ECC-Enabled Device Setup*.

Those already familiar with ZigBee security concepts can jump to section 4, Implementing Security.

2 Concepts

2.1 Network Layer Security

This section describes how ZigBee implements security at the network layer, which applies to standard security. Network security provides security independent of the applications that may be running on a ZigBee node. The application running on the coordinator can only decide whether or not network security will be used when forming the network. Afterwards, if network security is being used, then it is always used. The application has no ability to turn it off or send packets unencrypted. For application-controlled security, see section 2.3, APS Layer Security.

2.1.1 The Network Key

Network security uses a network-wide key for encryption and decryption. All devices that are authorized to join the network have a copy of the key and use it to encrypt and decrypt all network messages. The network key also has a sequence number associated with it to identify a particular instance of the key. When the network key is updated, the sequence number is incremented to allow devices to identify which instance of the network key has been used to secure the packet data. The sequence number ranges from 0 to 255. When the sequence number reaches 255, it wraps back to 0.

Note: All ZigBee keys are 128-bits in length.

All devices that are part of a secured ZigBee network must have a copy of the network key.

2.1.2 Hop-by-Hop Security

It is important to note that network security in ZigBee is done on a hop-by-hop basis. Each router that relays an encrypted packet first verifies that it is a valid encrypted packet before any more processing is done. A router authenticates the packet by executing the ZigBee decryption mechanism and verifying the packet integrity. It then re-encrypts the packet with its own network parameters (such as source address and frame counter) before sending the message to the next hop. Without this protection, an attacker could replay a message into the network that would be routed through several devices, thereby consuming network resources. Using hop-by-hop security allows a router to block attempts to inject bad traffic into the network.

2.1.3 Packet Security

A packet secured at the network layer is composed of the elements shown in Figure 2.

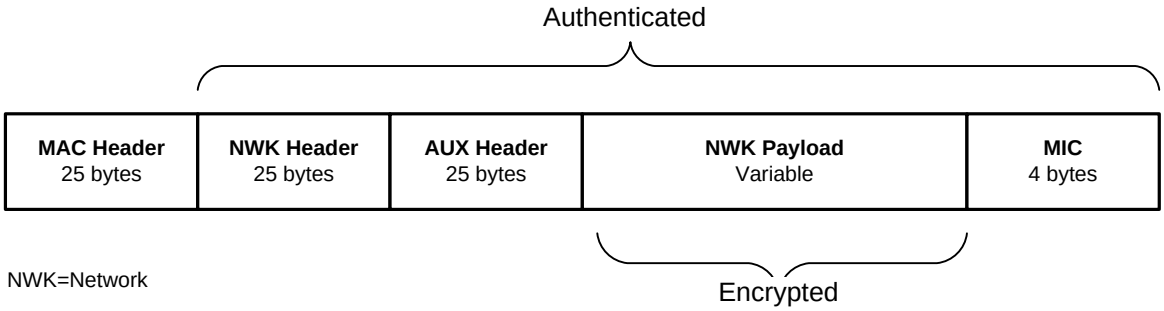

Figure 2. Anatomy of a Packet Secured at the Network Layer

2.1.4 Auxiliary Header

The auxiliary header contains data about the security of the packet that a receiving node uses to correctly authenticate and decrypt the packet. This data includes the type of key used, the sequence number (if it is the network key), the IEEE address of the device that secured the data, and the frame counter.

2.1.5 Authentication and Encryption

ZigBee uses a 128-bit symmetric key to encrypt all transmissions at the network layer using AES-128. The network and auxiliary headers are sent in the clear but authenticated, while the network payload is authenticated and encrypted. AES-128 is used to create a hash of the entire network portion of the message (header and payload), which is appended to the end of the message. This hash is known as the Message Integrity Code (MIC) and is used to authenticate the message by insuring it has not been modified. A receiving device hashes the message and verifies the calculated MIC against the value appended to the message. Alterations to the message invalidate the MIC and the receiving node will discard the message entirely.

Note: ZigBee uses a 4-byte MIC.

2.1.6 The Network Security Frame Counter

A frame counter is included in the auxiliary headers as a means of protecting against replay attacks. All devices maintain a list of their neighbor's and children's frame counters. Every time a device sends a packet, it increments the frame counter. A receiving device verifies that the frame counter of the sending device has increased from the last value that it saw. If it has not increased, the packet is silently discarded. If the receiving device is not the intended network destination, the packet is decrypted and modified to include the routing device's frame counter. The packet is then re-encrypted and sent along to the next hop.

The frame counter is 32 bits and may not wrap to zero. The network key can be updated before the frame counter reaches its maximum value. When that occurs, the sequence number is reset to zero to reflect the use of a different network key.

2.1.7 Unencrypted Network Data

If network security is being used, all packets are secured. The only exception is during joining, when devices do not yet have the network key. In that case a joining device's messages are relayed through its parent until it is fully

joined and authenticated. Any other messages that are received without network layer security are silently discarded.

2.2 Trust Center

Authentication in a secure network is controlled by means of a central authority known as a trust center. All devices entering the network are temporarily joined to the network until the trust center is contacted and decides whether or not to allow the new device into the network. The parent of the newly joined device acts as a relay between the trust center and the joining device. Only authentication messages can be sent to or from the device until it is fully joined and authenticated.

The trust center has the option of doing one of three things when a device joins:

- Send a copy of the current network key, which the parent relays to the joining device.
- Send a dummy network key that is encrypted with the real network key, which is relayed by the parent to the joining device. A joining device that is able to decrypt and read the message knows that it already has the current network key.
- Send the parent a command to remove the device from the network, thereby disallowing it from joining.

Once the node has the network key, it is considered fully joined and authenticated, and may communicate with any device on the network.

Standard security also relies on a trust center to authenticate devices joining the network. The trust center has the added responsibilities of distributing and managing trust center link keys and responding to requests for application link keys.

2.3 APS Layer Security

This section describes how ZigBee implements security at the Application Support (APS) layer. This applies to standard security only. The use of application layer security is optional in the ZigBee stack profiles but may be required by ZigBee application profiles.

2.3.1 End-to-End Security

APS security is intended to provide a way to send messages securely in a network such that no other device can decrypt the data except the source and destination. This is different than network security, which provides only hop-by-hop security. In that case every device that is part of the network and hears the packet being relayed to its destination can decrypt it.

APS security uses a shared key that only the source and destination know about, thus providing end-to-end security.

Both APS layer and network layer encryption can be used to encrypt the contents of a message. In that case APS layer security is applied first, and then network layer security.

2.3.2 Link Keys

APS security uses a peer-to-peer key known as the link key. Both devices must have already established this key with one another before sending APS-secured data. There are two types of link keys: trust center link keys and application link keys.

Trust Center Link Keys

The trust center link key is a special link key in which one of the partner devices is the trust center. The stack uses this key to send and receive APS command messages to and from the trust center. The application may also use this key to send APS-encrypted data messages.

Standard security does not require trust center link keys, but devices may request one after joining. Silicon Labs highly recommends using trust center link keys. They are required for any device that wishes to rejoin a network of which it was previously a member.

Application Link Keys

Application link keys are shared keys that may be established between any two nodes in the network. Optionally, they may be used to add additional security to messages being sent to or from the application running on a node. Devices can have a different application link key for each device with which they communicate.

A device may preconfigure an application link key or request a link key between itself and another device. In the latter case it issues a request to the trust center encrypted with its trust center link key. The trust center acts as a trusted third party to both devices, so they can securely establish communications with one another. This is discussed further in section 3.2.3, Application Link Keys. The process for establishing an application link key is illustrated in Figure 3.

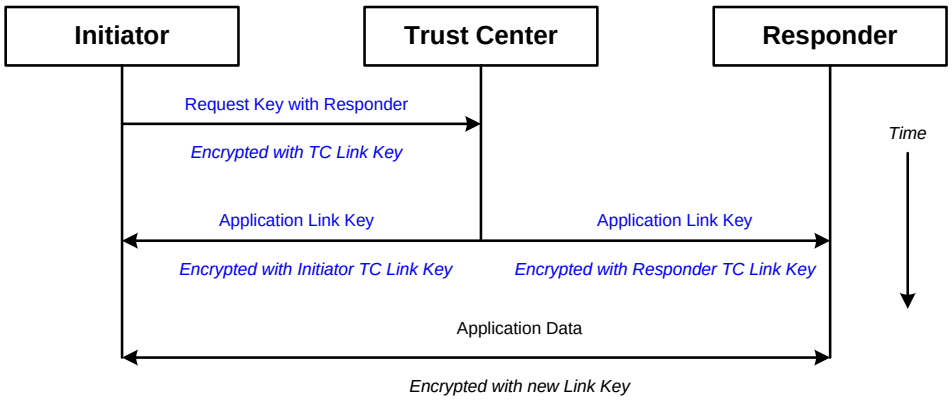

Figure 3. Establishing an Application Key

A packet secured at the APS Layer is composed of the elements shown in Figure 4.

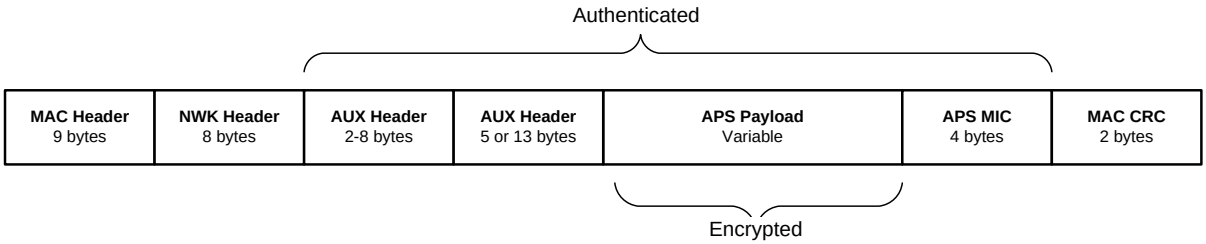

Figure 4. APS Packet Security

2.3.3 Unencrypted APS Data

APS layer security operates independently of network layer security. It is required for certain security messages (APS commands) sent to and from the trust center by the ZigBee stack.

Unlike network security, APS security for application messages is optional. Application messages are not automatically encrypted at the APS layer and are not ignored on the receiving side if they do not have APS encryption. Individual applications may choose whether to accept or reject messages that do not have APS layer security.

3 Standard Security

3.1 Overview

Standard security, introduced in the ZigBee 2007 specification along with ZigBee PRO, is the security model being used in all of the existing and developing ZigBee application profiles. It is the only security model (other than no security, which may be used for testing purposes only) supported by the EmberZNet stack libraries. It is only available to ZigBee PRO devices, but is backward-compatible with the “residential security” model defined in the legacy ZigBee 2006 specification and allows ZigBee 2006 devices to communicate securely on a ZigBee PRO network as end devices.

Standard security uses network keys and link keys to encrypt data at the network and application layers, respectively. The application support (APS) layer security allows the trust center to securely transport the network key to joining or rejoining nodes, and it optionally allows applications to add additional security to their messages. Network (NWK) layer security is used to secure all traffic sent on a ZigBee network, with the exception of basic MAC layer communication such as association, data requests (polling), and MAC ACKs.

One of the other significant features in standard security is the addition of end-to-end security at the application layer to supplement the network-wide security (see section 2.3, APS Layer Security). This allows individual nodes to establish secure communications that even other joined nodes with the network key cannot compromise. The trust center device takes on the additional responsibilities of helping establish this end-to-end security, managing key updates to individual nodes, and dictating network-wide security policies to which devices must adhere.

The benefits of standard security include the following:

- Link Keys — Link keys are used to create secure communications with another device regardless of network security. They are primarily used by the trust center to uniquely identify a device and send it secure data. With it, the trust center can send a message and be assured no other device can decrypt the message.

  Link keys can be used in certain cases where network encryption cannot, such as securing a message containing the network key. The trust center can be assured that only a node with the correct link key can decrypt and extract the network key.
- Rejoining — When a device rejoins the network, it may or may not have the current network key. Using standard security, a device can rejoin and receive an updated network key.

Because standard security is backward-compatible with residential security (ZigBee 2006 devices), the optional features present in standard security may not be supported by all devices on the network.

Note: ZigBee uses a key of all zeroes as a special indicator in security transactions. Therefore, it is not a valid network key. Additionally, the EmberZNet PRO stack reserves a key of all FF bytes as a special value, which cannot be used when setting up keys.

3.2 Use of Keys in Standard Security

Standard security defines different keys used for securing data in different ways. All keys are 128-bit symmetric and may or may not be used for encrypting/decrypting packets.

3.2.1 Network Key

This is the network-wide key used to secure transmissions at the network layer. Standard security requires the use of a shared network key among all devices in the network. The trust center may periodically update and switch to a new network key. The trust center first broadcasts a new network key encrypted with the old network key. Later it tells all devices to switch to the new network key. The new network key has a sequence number that is one higher than the last sequence number.

3.2.2 Trust Center Link Key

This key (known simply as the link key) is used for secure end-to-end communications between two nodes, one of which is the trust center. The trust center link key is used in these cases:

1. Encrypting the initial transfer of the network key to a joining node.
2. Encrypting an updated copy of the network key to a rejoining node that does not have the current network key.
3. Routers sending or receiving APS security messages to or from the trust center. These may be updates informing the trust center of a joining or rejoining node, or a command sent by the trust center to a router to perform some security function.
4. Application unicast messages that enable APS encryption, where either the sending or receiving device is the trust center.

The trust center has the option of deciding how to manage the trust center link keys. It may choose unique keys for each device in the network, keys derived from a common piece of shared data (the IEEE address of the device), or a global key that is the same for all devices in the network. Trust center link keys may also be negotiated at the application layer using a key establishment protocol like certificate-based key establishment (CBKE).

3.2.3 Application Link Keys

Standard security supports devices establishing application link keys with other devices. These keys are separate from the trust center link key and not required for normal operation. They are used for APS-level encryption between two devices in the network, neither of which is the trust center.

Application link keys must be established separately from the trust center link key. Devices may not establish an application link key with the trust center. However the trust center link key can be used to APS-encrypt application messages to the trust center, or from the trust center to a device on the network.

Application link keys can be established in one of two ways:

1. Manual configuration by the application specifying the key associated with a destination device.
2. Through a request that the trust center generate a key and send it to both devices.

The application can manually configure a key by calling into the stack and setting one up. The partner device must also configure the application link key and negotiate with the other device when they can start using that key.

Application link keys can also be established using the trust center. The Ember Stack supports two methods for this. The first is the ZigBee standard method, discussed earlier in section Link Keys and illustrated in Figure 3, where one device requests an application link key with another device by contacting the trust center. The trust center then immediately responds and sends a randomly generated application link key back to the requesting device and to the partner device. The drawback with having only one device request a key is that the other device may be asleep, offline, or have insufficient capacity to hold another key.

The second method, shown in Figure 5, is **not standardized in ZigBee** and will be non-interoperable with other vendors' devices. It also requires that all Ember devices in the exchange are configured to use this method, including the trust center. It is more reliable in that it helps ensure that the partner device is online and able to receive an application link key. In this case, both devices must request an application link key from the trust center. The trust center stores the first request for an application link key for a period of time defined by the trust center application. During that time, the partner must send in its own application link key request with the first device as its partner. If that occurs, then the trust center generates a random application link key and sends it back to both devices. Requiring both devices to request an application link key greatly reduces the chance that a device or its partner will not receive the key.

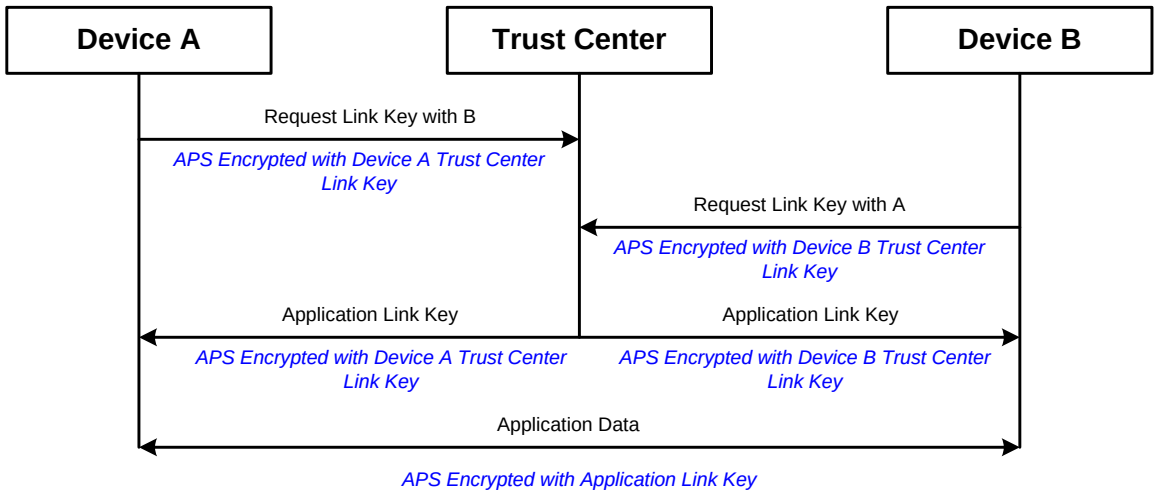

Figure 5. Requesting an Application Link Key with Another Device on the Network

EmberZNet PRO supports a configurable table for storing application link keys. See section 4.4, Common Security Configurations, for more information.

3.3 Joining a Network

A device initiates the process of joining a ZigBee standard security network by first using MAC association to join to a suitable parent device. If the association is successful, the device is joined but unauthenticated, as it does not possess the network key.

After sending the success response to the MAC association request, the router sends the trust center an Update Device message indicating that a new node wishes to join a ZigBee network. The trust center can then decide whether or not to allow the device to join. If the device is not allowed to join, a Remove Device request is sent to the parent, as shown in Figure 6. If the device is allowed to join, the trust center's behavior depends upon whether the device has a preconfigured link key.

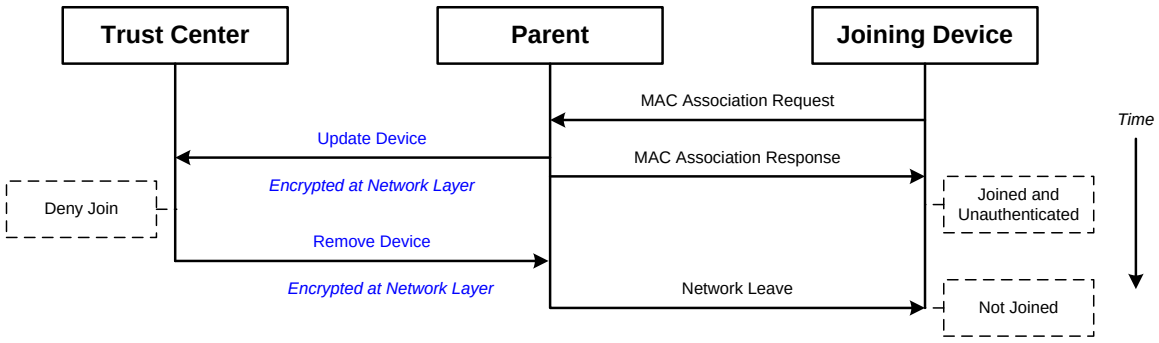

Figure 6. A Device that is Denied Access to Join the Network

3.3.1 Preconfigured Link Keys

The trust center dictates the policy of how to handle new devices and determines whether a device should have a preconfigured link key. If a new device does not have a preconfigured link key, it will be unable to join the network.

The trust center has the option of choosing how it assigns link keys to each device. It could use a single link key for all devices, a key derived from a bit of shared data (such as the joining node's EUI64 Address), or unique, randomly generated keys for each device. See Figure 7 for the joining process with a preconfigured key.

To allow a device onto the network, the trust center transmits the network key encrypted with the device's preconfigured link key.

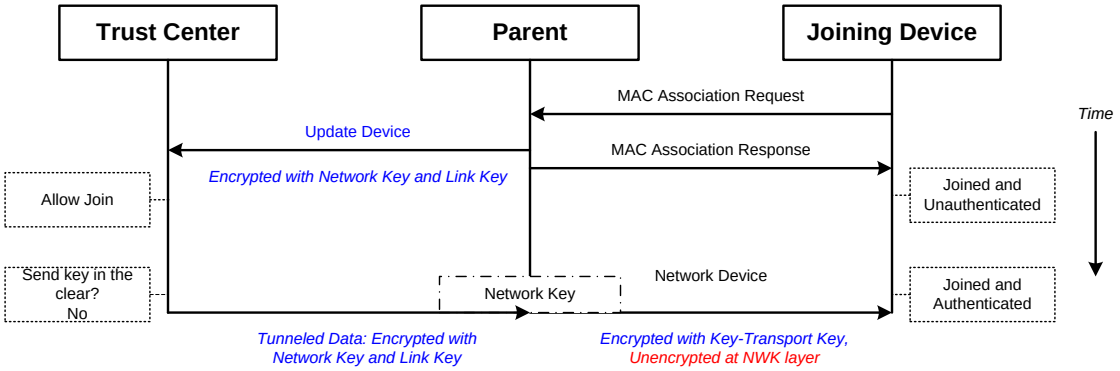

Figure 7. Joining Using a Preconfigured Trust Center Link Key

3.3.2 No Preconfigured Keys

The trust center may also allow devices onto the network that do not have a preconfigured link key. To do this, it must transmit the network key in the clear. This represents a security risk but it may be acceptable, depending on the application.

3.3.3 Requesting a Link Key

One of the advantages with standard security is that, even if a preconfigured link key is not required for joining (such as when the network key is sent in the clear), a device has the option of requesting one from the trust center. This is done after the device has received the network key, as shown in Figure 8.

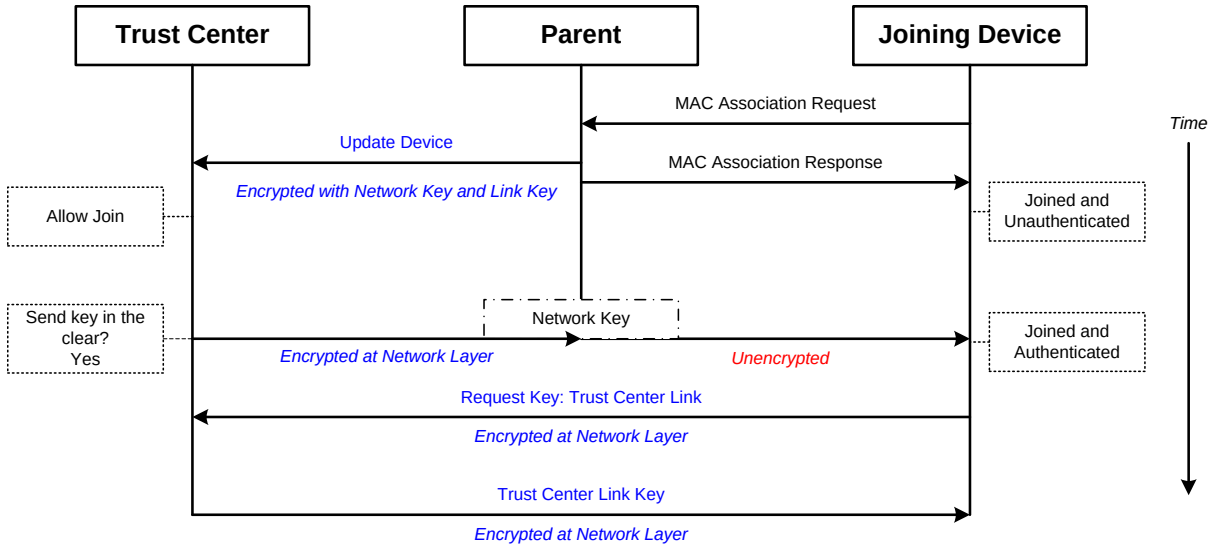

Figure 8. Joining Using a Preconfigured Trust Center Link Key

A trust center link key allows a device to rejoin even if it no longer has the current network key. This might happen if it missed a key update.

3.4 Network Key Updates

The network key encrypts all transmissions at the network layer. As a result, a local device constantly increases its local network key frame counter. Before any device in the network reaches a frame counter of all F's, the trust center should update the network key. Since it is not possible for the trust center to know the frame counter value of every device in the network at any given time, or even to inspect the frame counters of incoming messages, an approach that relies on specific frame counter thresholds is not practical. Thus, a preventative maintenance approach relying on periodic updates at long intervals, similar to what is described below, is recommended.

The recommended model is for the trust center to periodically update the network key to help minimize the risk associated with a particular instance of the network key being compromised. This helps to ensure that a device that has left a secured ZigBee Network is not able to rejoin later.

Key updates are broadcast by the trust center throughout the network, encrypted using the current network key. Devices that hear the broadcast do not immediately use the key, but simply store it. Later, a Key Switch is broadcast by the trust center to tell all nodes to start using the new key.

At a minimum, the trust center should allow adequate time (approximately 9 seconds) for the broadcast of the new key to propagate throughout the network before switching. In addition, a trust center must keep in mind that sleeping end devices may miss the initial broadcast unless they poll frequently.

It is possible that any device may miss a key update. This may happen because it was sleeping, was powered off, or dropped off the network for an extended period of time. If this occurs, a device may try to perform an unsecured rejoin. The trust center can then decide whether to allow the node back on the network.

The EmberZNet PRO stack can detect the condition where an encrypted packet arrives secured with a newer network security key. It will automatically perform an unsecured rejoin to its current network to attempt to acquire the latest network key.

3.5 Network Rejoin

Rejoining is a way for a node to reconnect to a network of which it was previously part. Rejoining is necessary in two different circumstances:

1. Mobile or sleepy devices that may no longer be able to communicate with their parent.
2. Devices that have missed the network key update and need an updated copy of the network key.
3. Devices that have missed a PAN ID update and need to discover the network’s new PAN ID.

When a device tries to rejoin, it may or may not have the current network key. Without the correct network key, the device's request to rejoin is silently ignored by nearby routers.

Therefore, a device has two choices when rejoining: a secured rejoin or an unsecured rejoin. Note that neither of these rejoin cases requires the MAC Permit Association (also known as “permit joining”) flag to be set on any devices in the target network. A router/coordinator device will always accept a NWK layer Rejoin command that is either unsecured or secured with the active network key. It is the trust center’s responsibility to grant or deny access once it is notified of the rejoining activity.

If the device has neither the current network key nor a trust center link key, it will have to perform a join.

3.5.1 Secured Rejoining

A secured rejoin is the easier case and a device seeking to rejoin the network should try this method first. If it has the current network key, the device will be able to communicate on the network again very quickly. A secured rejoin is only necessary when a sleepy or mobile end device has lost its parent.

As illustrated in Figure 9, the device sends its rejoin request encrypted with its copy of the network key. If a router is nearby and is using the same network key, the rejoin response is sent back to the device encrypted. The device is now joined and authenticated on the network again. The parent that answered the rejoin request informs the trust center that the device rejoined, but no further action must be taken by the trust center.

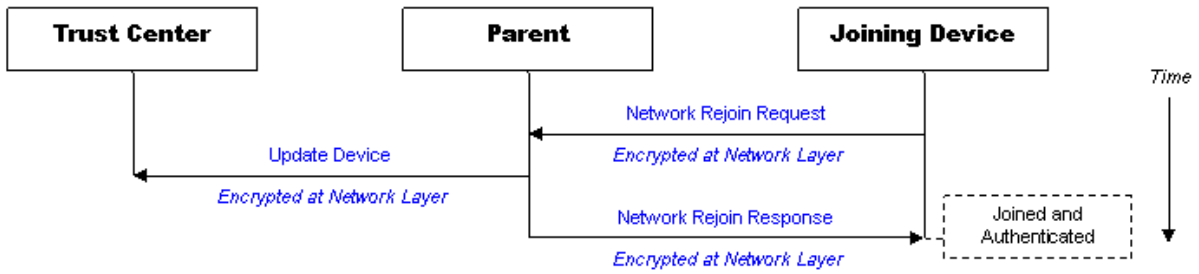

Figure 9. Secured Rejoin

If the secured rejoin fails and the device is using standard security, the application can try an unsecured rejoin.

3.5.2 Unsecured Rejoining

An unsecured rejoin is necessary when neighboring devices have switched to a new network key and no longer use the same network key as the rejoining device. To succeed in the unsecured rejoin, the device must have a trust center link key. The device sends the rejoin request unencrypted. A nearby router accepts the unencrypted rejoin request and responds to the device, allowing it to transition to the joined and unauthenticated state.

As illustrated in Figure 10, the parent of the rejoining device sends an Update Device message to the trust center, informing it of the unsecured rejoin. The trust center has two choices: deny or accept the rejoin. If it accepts the rejoin, it must send an updated network key to the device. However, it secures this message using that device's trust center link key. The message is sent to the parent of the rejoining device encrypted at both the network and APS layers. The parent then relays this message without network encryption to the rejoining device. Once it has the network key, it will be in the joined and authenticated state and can communicate on the network again.

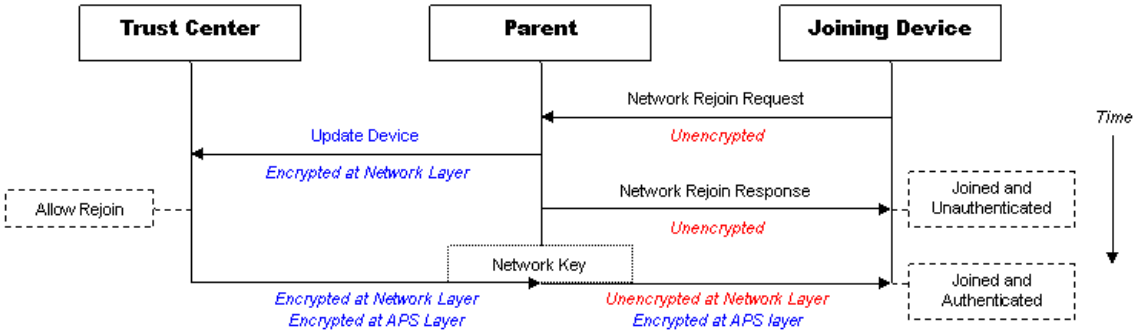

Figure 10. Unsecured Rejoin

3.6 Trust Center Decision Process Summary

Figure 11 illustrates the decision tree for the trust center when a device joins the network. The parent of a joining or rejoining device sends an Update Device APS command to the trust center, indicating the event has taken place. The trust center application decides what to do based on that information. This figure describes the behavior for a ZigBee PRO device joining a ZigBee PRO network using standard security.

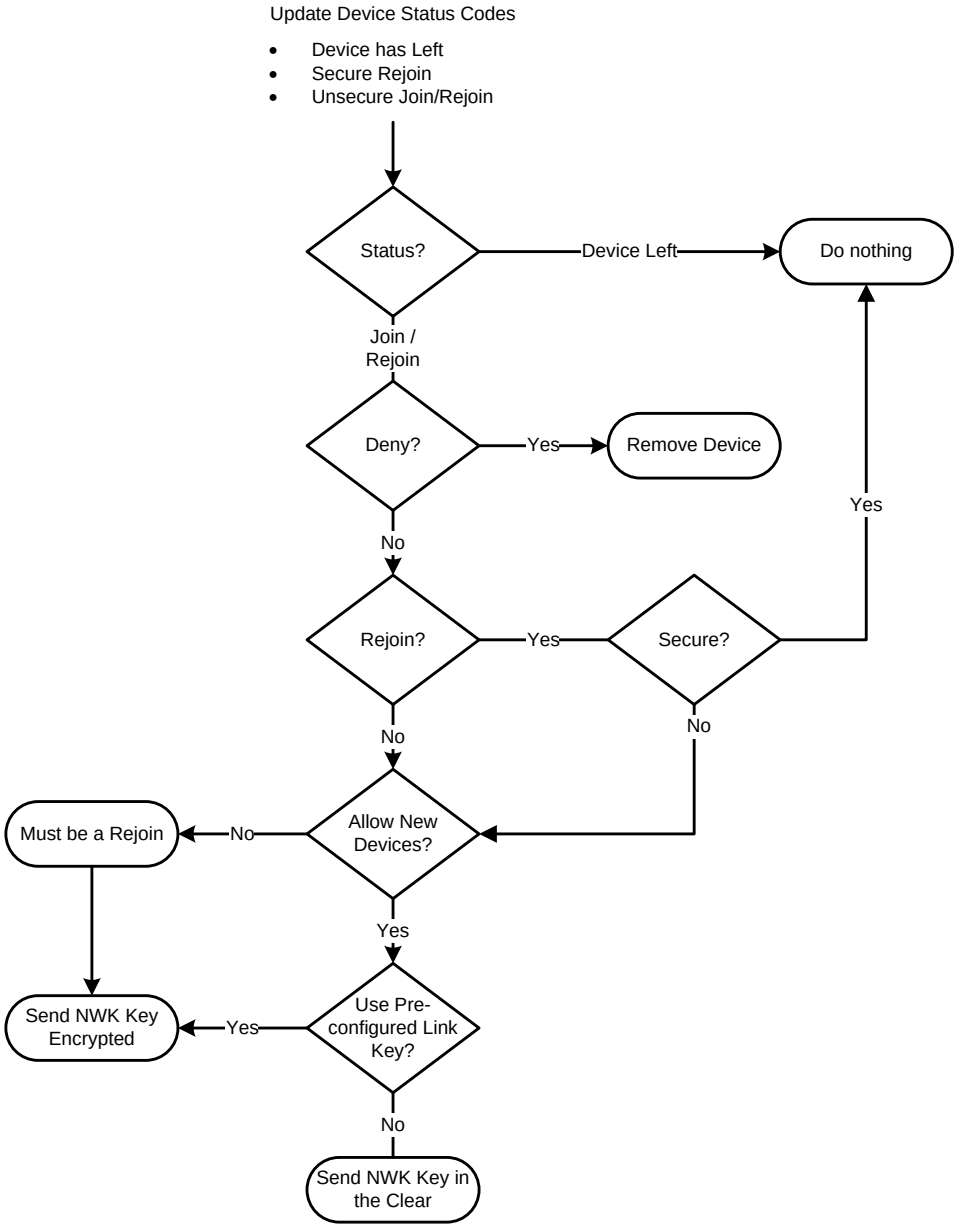

Figure 11. Decision Process for the Trust Center

The trust center can decide whether or not to allow devices into a ZigBee network and whether or not to send the key in-the-clear. The trust center's decision can be made based on any number of additional factors, such as a user event (button press), a time-based condition, IEEE address of the joining device, or some other condition (such as, the network is being commissioned).

When new devices join, the trust center decides whether the device should have a preconfigured key. The joining devices have no ability to inform the trust center through the ZigBee protocol about whether or not they have a preconfigured key.

3.6.1 Standard Security Without a Trust Center

Normally a joining device is authenticated by the trust center through its parent. This is advantageous, as it allows one device to act as a gatekeeper and authenticate all devices that want to join the network. Security messages are relayed to the joining device through its parent until it becomes joined and authenticated.

However, this means that all routers must have a route to the trust center and vice versa. When applications are being developed or when commissioning a network, the trust center may not be reachable, and thus devices cannot join.

The EmberZNet PRO stack allows a network to use standard security features without a trust center. This is known as distributed trust center mode, and it is not ZigBee-compliant. This mode has the advantage of permitting devices to join without requiring the parent node to send information to the trust center and await the response. In this mode, all routers mimic the behavior of a trust center by sending the security data directly to the joining node. Each router individually decides whether or not to let the device onto the network. This mode is useful to allow commissioning of a complete network and then establishment of a trust center for security.

Note: Before EmberZNet 3.2, "Distributed Trust Center Mode" was known as "No Trust Center Mode."

In this mode, all devices use a single trust center link key that may be preconfigured or not. If using a preconfigured link key, the router sends the network key encrypted to the joining device. If not using a preconfigured link key, the router sends the network key in the clear.

All devices inherit the distributed trust center setting from their parent when they join and also operate in that mode. Thus, only the device that forms the network (the coordinator) needs to be set up to run in distributed trust center mode.

Note: Your application cannot be certified as ZigBee-compliant without implementing a trust center.

3.6.2 Changing a Network to Use a Trust Center

A standard security network without a trust center also has the potential to later operate with a trust center. A network can be commissioned without a trust center and later transition to use a trust center, once all the initial setup is finished.

The device that wishes to become the trust center must be the coordinator. That device informs the network through a network key update that it is the new trust center.

3.7 Additional Requirements for a Trust Center

To function correctly in a ZigBee PRO network, a trust center also requires that:

1. The trust center application must act as a concentrator (either high or low RAM).
2. The trust center application must have support for source routing. It must record the source routes and properly handle requests by the stack for a particular source route.
3. The trust center application must use an address cache for security, in order to maintain a mapping of IEEE address to short ID.

Failure to satisfy all of the above requirements may result in failures when joining/rejoining devices to the network across multiple hops (through a target node that is neither the trust center nor one of its neighboring routers.)

3.7.1 Trust Center as a Concentrator

The trust center must act as a concentrator because ZigBee PRO Security requires two-way routes to and from the trust center in order to transmit all the security messages necessary to transition a device to the joined and authenticated state.

Routers running the EmberZNet PRO stack automatically add a route to the trust center through their parent (the device they joined to) immediately after they become joined and authenticated. This route assumes that the trust center is acting as a low RAM concentrator.

The trust center should periodically broadcast the many-to-one route message, so that all routers update their routing tables and repair broken routes to the trust center. This also allows it to notify routers if it is acting as a high RAM concentrator, thereby updating the default route.

3.7.2 Trust Center and Source Routing

The trust center must have support for source routing in the application. It should record the routes of incoming messages and store them in its own table. If the trust center is acting as a high RAM concentrator, it must keep track of all source routes.
If the trust center is acting as a low RAM concentrator, then only the last couple of source routes must be recorded. The minimum number of entries in the source route table should be sized to support the maximum number of simultaneous security events that may occur at one time. These security events include rejoins, joins, and leaves.

In addition to storing the source routes, the trust center must also implement the proper hooks to respond to requests by the stack for a particular source route. Silicon Labs provides a source route library, which manages a source route table, and works with a trust center.

Note: For EZSP host-based designs, the source route table on the host cannot be used for routing security messages sent to devices joining or rejoining the network. This is because the APS security transactions must be handled by the stack (in the network coprocessor) without relying on application level interaction (at the host.)

3.7.3 Trust Center Address Cache

In order to properly decrypt APS-encrypted messages, the trust center must maintain a mapping of IEEE address to short ID.

For a high RAM concentrator, the trust center must keep track of all devices in the network.

For a low RAM concentrator, the trust center need only keep track of a couple of entries at a time and may overwrite old entries as needed. The size of the cache should be equal to the maximum number of simultaneous security events that can occur at one time. These security events include rejoins, joins, and leaves.

Silicon Labs provides sample code for implementing the trust center address cache mechanism on SoC platforms (See app/util/security/security-address-cache.c). The Ember EZSP network coprocessor binaries contain the security address cache feature as part of the firmware, but the host application must enable this feature by setting the Trust Center Address Cache Size to a value greater than 0. (Refer to document UG100, *EZSP Reference Guide*, for details on how to do this in your EZSP host implementation.)

4 Implementing Security

Security is implemented through stack software and message packet configuration. Security configuration in a typical application is often assisted by the Ember security utilities module (found in app/util/security within the EmberZNet PRO stack installation) and the Ember application framework (found in the app/framework directory of the EmberZNet PRO stack installation).

If you are implementing a ZigBee-compliant application design based on the ZigBee Home Automation or ZigBee Smart Energy application profiles, please refer to sections 4.4.10, Security Setup for ZigBee Home Automation (ZHA) Devices, and 4.4.11, Security Setup for ZigBee Smart Energy (ZSE) Devices, as well as section 5, ZigBee Smart Energy (ZSE) Security, which focuses exclusively on ZSE security. These implementations generally rely on the Ember application framework to configure security settings according to the chosen profile.

Note: Complete documentation for Ember application framework can be found in document UG102, *Application Framework Developer Guide*. Complete API documentation for the Ember security utilities module and the EmberZNet PRO stack itself can be found in the EmberZNet PRO API Reference provided for your target platform. Refer to the specific versions of these documents included in your EmberZNet PRO software installation to ensure the information is accurate for your software version.

4.1 Turning Security On or Off

In order to use security in the application, the following must be set in the application's configuration header.

```
#define EMBER_SECURITY_LEVEL 5
```

Security level 5 is the only level supported by ZigBee, and defines both authentication and encryption at the network layer. Silicon Labs also supports security level 0 (no security). To enable this, set the security level as follows in the application's configuration header.

```
#define EMBER_SECURITY_LEVEL 0
```

Disabling security in the application is not ZigBee-compliant and may therefore only be used with networks for which EMBER_STACK_PROFILE is defined as 0 (custom stack feature set rather than ZigBee PRO).

4.1.1 EZSP NCP Configuration

On EZSP Network Coprocessor [NCP] platforms, security is turned off by default (level 0). If the host application desires network security, then it must turn on security at the stack level by setting the appropriate configuration value in the EZSP firmware, as shown in the following example:

```
ezspSetConfigurationValue(EZSP_CONFIG_SECURITY_LEVEL, 5);
```

After a reboot, the NCP falls back to the default configuration (with security disabled). Therefore, an application that wants to run with security must set the security level configuration value after every reboot. This is generally accomplished with an NCP initialization sequence in which the host application completes the EZSP Version transaction and then pushes all desired configuration settings to the NCP before resuming network operation as part of the NCP reset/initialization process.

4.2 Security for Forming and Joining a Network

Devices forming or joining a ZigBee PRO network using standard security must call `emberSetInitialSecurityState(…)` before calling `emberFormNetwork(…)` or `emberJoinNetwork(…)`.

```
boolean emberSetInitialSecurityState(EmberInitialSecurityState* state)
```

All initial security parameters are set through a single call to `emberSetInitialSecurityState(…)`. The function takes a data structure containing the settings and (optionally) the keys. It is used only to set up security before forming or joining. Once security is set up and a device is joined into the network, it persistently stores those settings. Security parameters cannot be changed unless the device leaves.

On startup an application should always call `emberNetworkInit(…)` first before calling `emberSetInitialSecurityState(…)`. If the device is already joined in the network

(EMBER_JOINED_NETWORK or EMBER_JOINED_NETWORK_NO_PARENT) then it is not necessary to call emberSetInitialSecurityState(…).

Upon a successful call to emberSetInitialSecurityState(…) the outgoing and incoming APS and NWK frame counters are reset to zero.

4.2.1 The Initial Security State Structure

The following data structure is used by emberSetInitialSecurityState(...) and enumerates the security parameters that are used when joining:

```
typedef struct {
  /** This bitmask enumerates which security features should be used, as well
      as the presence of valid data within other elements of the
      ::EmberInitialSecurityState data structure.  For more details see the
      ::EmberInitialSecurityBitmask. */
  int16u bitmask;
  /** This is the pre-configured key that can used by devices when joining the
   *  network if the Trust Center does not send the initial security data
   *  in-the-clear.
   *  For the Trust Center, it will be the global link key and <b>must</b> be set
   *  regardless of whether joining devices are expected to have a pre-configured
   *  Link Key.
   *  This parameter will only be used if the EmberInitialSecurityState::bitmask
   *  sets the bit indicating ::EMBER_HAVE_PRECONFIGURED_KEY*/
  EmberKeyData preconfiguredKey;
  /** This is the Network Key used when initially forming the network.
   *  This must be set on the Trust Center.  It is not needed for devices
   *  joining the network.  This parameter will only be used if the
   *  EmberInitialSecurityState::bitmask sets the bit indicating
   *  ::EMBER_HAVE_NETWORK_KEY.  */
  EmberKeyData networkKey;
  /** This is the sequence number associated with the network key.  It must
   *  be set if the Network Key is set.  It is used to indicate a particular
   *  of the network key for updating and swtiching.  This parameter will
   *  only be used if the ::EMBER_HAVE_NETWORK_KEY is set. Generally it should
   *  be set to 0 when forming the network; joining devices can ignore
   *  this value.  */
  int8u networkKeySequenceNumber;
  /** This is the long address of the trust center on the network that will
   *  be joined.  It is usually NOT set prior to joining the network and
   *  instead it is learned during the joining message exchange.  This field
   *  is only examined if ::EMBER_HAVE_TRUST_CENTER_EUI64 is set in the
   *  EmberInitialSecurityState::bitmask.  Most devices should clear that
   *  bit and leave this field alone.  This field must be set when using
   *  commissioning mode.  It is required to be in little-endian format. */
  EmberEUI64 preconfiguredTrustCenterEui64;
} EmberInitialSecurityState;
```

4.2.2 Initial Security Bitmask

Table 1 describes the different settings used for security and the devices that may set them:

Table 1. Initial Security Bitmask

Bits	Name	May be set by:	Description
1	EMBER_DISTRIBUTED_TRUST_ CENTER_MODE	Device that forms network	Controls whether the device creates a network operating with (0) or without (1) a trust center. All devices that join the network inherit this setting from their parent.
2	EMBER_GLOBAL_LINK_KEY	Trust center	Controls whether the trust center is using the same link key for all devices (1), or separate link keys for each device (0). Must be set when operating in distributed trust center mode.
3	EMBER_PRECONFIGURED_ NETWORK_KEY_MODE	Trust center	Controls whether the trust center uses a ZigBee 2006 compatibility mode for end devices with a preconfigured network key. ZigBee 2006 devices with a preconfigured network key require that a dummy network key be sent to them to indicate they have the right security key. This enables (1) or disables (0) that behavior. ZigBee PRO devices are unaffected by this.
4-5	Reserved		
6	EMBER_HAVE_TRUST_CENTER_ EUI64	Joining devices	Determines whether the device has set a value in the preconfiguredTrustCenterEui64 field in the EmberInitalSecurityState structure. If the value is set, that field is taken as the EUI64 of the trust center in the network. Normally this field should NOT be set, because a joining device learns the EUI64 of the trust center when it receives the network key. However, a device that is being commissioned to join an existing network without sending any over-the-air messages must also set this bit and populate the field appropriately.
7,2	EMBER_TRUST_CENTER_USES_ HASHED_LINK_KEY	Trust center	Controls whether the trust center creates semi-unique link keys for each device in the network by hashing the preconfigured key with the IEEE address of the device to obtain the real link key. This is one method by which the trust center can ensure that devices do not share the same trust center link key throughout the network.
8	EMBER_HAVE_PRECONFIGURED _KEY	Forming or joining devices	Controls whether or not valid data is in the preconfiguredKey element of the EmberInitialSecurityState structure. If set (1), the stack records the key in persistent storage. If not set (0), that parameter is ignored. Note that this preconfigured key represents the trust center link key, which, for the trust center, will be treated as the global link key if EMBER_GLOBAL_LINK_KEY is set in the EmberInitialSecurityBitmask. Must be set for the device that forms the network, and is separate from the trust center decision (as each node joins) to use preconfigured keys. For joining devices, controls whether or not the device has a preconfigured key.
9	EMBER_HAVE_NETWORK_KEY	Forming device	Controls whether or not valid data is in the networkKey element of the EmberInitialSecurityState structure. If set (1), the stack records the key in persistent storage. If not set (0), that parameter is ignored. Must be set before forming the network. It is not needed for joining.
10	EMBER_GET_LINK_KEY_ WHEN_JOINING	Joining devices	Controls whether the joining device will request a trust center link key after it receives the network key. If set (1), the joining is not successful until a request is sent and a response is received. If unset (0), the device does not request a link key. This is only necessary if the device does not have a preconfigured trust center link key.

Bits	Name	May be set by:	Description
11	EMBER_REQUIRED_ENCRYPTED_KEY	Joining devices	Controls whether a joining device with a preconfigured link key accepts a network key sent in the clear. If set (1), a network key sent in the clear is rejected. If unset (0), the device accepts either an encrypted network key or a network key sent in the clear. It is recommended to set this if a preconfigured key is being used.
12	EMBER_NO_FRAME_COUNTER_RESET	Forming or joining devices	Denotes whether the device should not reset its outgoing NWK and APS frame counters before joining or forming the network. Normally all frame counters are reset to maximize the number of available outgoing frame counters that may be used. However, if this bit is set, that behavior is overridden and the device uses whatever previous value was stored in the tokens. This is used when a device will join a network that it was previously part of, but is not using the API emberRejoinNetwork().
13	EMBER_GET_PRECONFIGURED_KEY_FROM_INSTALL_CODE	Joining devices	Denotes whether the device should get its Preconfigured link key from a Smart Energy installation code set in a token. A Smart Energy-compliant device must set this bit and preprogram an installation code into its token area. If the token is not set or is invalid, the call to emberSetInitialSecurityState()fails.
14-15	Reserved		

4.3 Security Keys

The trust center (or coordinator, for distributed trust center networks) must set both the preconfigured key, which will be the trust center link key, and the network key in the `EmberInitialSecurityState` structure. Joining devices should not set the network key, and may optionally set the preconfigured key in the structure.

Note that if a joining device is using a Smart Energy installation code to derive its initial trust center link key through the AES MMO hash function (that is, `EMBER_GET_PRECONFIGURED_KEY_FROM_INSTALL_CODE` is set), then it is not necessary for a joining device to specifically provide this preconfigured key. In this case the preconfigured key is calculated and recorded by the stack for use when the device joins the network. The calculated link key is set in the `preconfiguredKey` value in the structure and returned to the caller when the `emberSetInitialSecurityState(…)` completes.

The network key may take on any value, except a value of all zeroes. An all-zero key is a special reserved value in ZigBee standard security. Attempts to call `emberSetInitialSecurityState(…)` with `EMBER_HAVE_NETWORK_KEY` and a network key of all zeroes will fail.

4.3.1 Frame Counters

Encrypted messages use frame counters to prevent messages from being replayed into the network. Frame counters are of two types: network frame counters and APS frame counters.

Network frame counters are used to keep track of messages encrypted at the network layer with the network key. All devices have a single outgoing network frame counter stored persistently in flash. The coordinator and routers keep track of all the incoming frame counters of their neighbors and children. End devices keep track of only their parent's incoming frame counter.

Network frame counters provide basic protection against replaying messages between two adjacent devices. However they have limited ability to protect against replayed messages to non-adjacent neighbors. Devices may be able to use APS encryption to protect against those attacks.

APS frame counters are used to keep track of messages encrypted at the APS layer with a link key. Since the link key is normally shared between only two devices, messages encrypted with that key have a higher level of security.

All devices have a single outgoing APS frame counter. All non-trust center devices store the incoming frame counters of the trust center link key and application link keys.

4.3.2 Outgoing Frame Counter Tokens

The outgoing frame counters for NWK and APS are stored in RAM and only written to flash periodically. After every 4,096 messages (0x1000) the local device's outgoing frame counter is written to flash. After a reboot, once the stack has been initialized and where at least one message has been queued up to be sent, the frame counter is rounded up to the next multiple of 4,096.

4.3.3 Replay Protections

Replay protection is dependent upon the storage of the incoming frame counter. Outgoing frame counters for a device are stored persistently in flash. However, incoming frame counters are stored only in RAM. After a reboot the incoming frame counter is reset to 0. In order to synchronize the frame counters for a pair of devices after reboot, a challenge response mechanism is needed. No such mechanism is currently supported by ZigBee PRO standard security.

The following types of replay protection are possible:

- **None:** Incoming frame counters are not checked. Application-specific means must be used to insure against replay attacks.
- **Weak:** Incoming frame counters are checked and maintained as long as the device is powered on and the device has the storage capacity. A reboot causes the device to forget the incoming frame counters and reset all of them to 0.
- **Strong:** Incoming frame counters are checked and maintained as long as the device is powered on. A device that reboots re-synchronizes the frame counter with its partner using a challenge-response mechanism. This is not available in the EmberZNet PRO stack libraries as it is not supported in standard security.

The trust center maintains a trust center link key with all devices on the network. Trust centers normally do not keep track of the incoming APS frame counters of every device on the network. Storing individual link keys and frame counters for every single device on the network requires a significant amount of flash and RAM and does not scale as the network grows. Therefore, the trust center has a number of options:

1. A global trust center link key
2. A hashed trust center link key
3. Unique trust center link keys for a small number of devices
4. A mix of the above possibilities.

Each of these options is discussed in the following sections.

4.3.4 Global Trust Center Link Key

A global trust center link key is the simplest option and the one adopted by the ZigBee Home Automation application profile (among others). Only one trust center link key is used for all devices on the network. Any messages using APS encryption are encrypted with that key. Preconfiguration of the trust center link key on joining nodes is made easier since the same key is used by all devices. The trust center does not keep track of the incoming frame counter for that key since multiple nodes are using that key and will increment their frame counters differently.

Although the trust center knows that the key is global to the network, joining devices do not. To those devices, the key is unique.

A global trust center link key is the only key supported for a distributed trust center mode network.

4.3.5 Hashed Trust Center Link Key

A hashed trust center link key allows the trust center to appear to be using completely unique link keys, but only store one key for all devices on the network. When the trust center attempts to APS encrypt or decrypt a message, it performs a special operation to derive the link key. The trust center uses a root key hashed with the IEEE Address of the device that is the recipient or sender of the message, in order to obtain the actual link key associated with a specific device.

The root key is a secret key known only to the trust center (and possibly a commissioning device). This key does not need to be distributed to joining nodes. The hashed value of the root keys is the only trust center link key known to the other nodes on the network.

The advantage of the hashed key is that all devices on the network are using a unique key to encrypt and decrypt messages, but the trust center only needs to store a single key. The trust center does not keep track of incoming APS frame counters for individual devices that are using a hashed link key.

The algorithm used to create the key is the hashed message authentication code (HMAC). The block cipher used for the HMAC algorithm is AES-128. The function to derive the key can be written as follows:

```
A = Little Endian IEEE Address of device A
R = Root Key
U = Unique Key

U = HMAC(A, R)
```

HMAC is an open and public standard created by the U.S. Government. More information can be found here:

http://csrc.nist.gov/groups/ST/toolkit/message_auth.html

Some disadvantages to using the hashed trust center link key method include the following:

- This method of key derivation is not standard among ZigBee PRO stack vendors and therefore is non-interoperable with networks where the trust center is not running the EmberZNet PRO stack.
- Generating semi-unique link keys for each device before it enters the network requires either that (1) the security administrator of the network generates these keys for all devices expected to participate and distributes these keys to manufacturers or installers of these devices somehow; or (2) the security administrator of the network provides the root key to device manufacturers and allows them to generate the HMAC-based link key themselves. The first option is difficult to organize in advance as the EUI64 of incoming devices is not usually well-known in advance of them entering the network. The second option increases burden on device manufacturers and greatly increases risk of compromising the root key.
- If the root key ever changes, all joining devices would need a different preconfigured key.

4.3.6 Unique Trust Center Link Keys

The trust center does have the option of using totally unique trust center link keys for a small set of devices in the network. This is the security configuration adopted by the ZigBee Smart Energy application profile. In this case the trust center will store the incoming frame counters associated with those link keys and thereby has replay protection for APS encrypted messages it receives from those devices.

Unique trust center link keys require the full link key security library (security-library-link-keys as opposed to security-library-link-keys-stub). The trust center must set up a key for the device before that device joins, regardless of whether the key is being sent in-the-clear or is preconfigured. This is accomplish using the `emberSetKeyTableEntry()` or `emberAddOrUpdateKeyTableEntry()`functions available in the stack API and in EZSP. Each entry in the table consumes both flash and RAM. For more information on the flash and RAM requirements see section 4.8, Link Keys Library.

Note that the EmberZNet PRO stack also supports the derivation of unique keys at runtime through an AES MMO hash algorithm computed over a Smart Energy installation code stored in non-volatile memory.

4.3.7 Mixed Link Keys

It is possible for the trust center to support a mix of link key types. The following are supported:

1. Unique trust center link keys for certain devices and a global link key for all others.
2. Unique trust center link keys for certain devices and a hashed link key for all others.

When the trust center tries to encrypt or decrypt a message using a link key, it first consults the link keys table for a unique trust center link key associated with the sending/receiving device. If none exists, then it falls back on the global or hashed link key.

This setup is advantageous as it supports having the incoming APS frame counter protection for certain keys with specific nodes, while scaling to support any number of other devices in the network by using a global or hashed trust center link key.

4.3.8 Summary of Replay Protection

Table 2 provides a summary of the replay protection provided under the different security configurations supported.

Table 2. Replay Protection Level

Layer	Device Type	Key Type	Incoming Frame Counter Protection
APS	Trust center	Global trust enter link key	None
		Hashed trust center link key	None
		Unique trust center link key	Weak
	Router or end device	Trust center link key	Weak
		Application link key	Weak
		Trust center link key derived using Smart Energy's Certificate-Based Key Exchange	Weak
NWK	Trust center, router or end device	Network key	Weak

Note: Routers or end devices treat the trust center link key as unique and therefore have some replay protection when receiving messages. Those devices do not know whether the trust center is using a global, hashed, or unique link key.

4.4 Common Security Configurations

Security has many different options and settings that can make it difficult to set up. Most of the decisions lie with the trust center. Setting security for non-trust center devices involves a much smaller number of options.

4.4.1 Joining Nodes

Joining nodes have only a couple decisions to make about security. The rest will be dictated by the network and or the trust center.

1. Will preconfigured link keys be used or will the key be sent in-the-clear?
2. Will application link keys be used to APS-encrypt data to non-trust center devices?

Preconfiguring the link key involves setting it on the device at any point before the device joins the ZigBee network. This could be done at manufacturing time by storing the value in the device, during installation of the node using some commissioning tool, or even through non-ZigBee wireless methods where the key is sent by a short range, low power radio transmission. Whatever method is used, the preconfigured key should be treated carefully to prevent it from being distributed beyond those devices that must possess it.

If the device joins a network operating without a trust center (distributed trust center mode) the device learns about this when it joins the network. It does not need to set anything up when joining the network since it has no control over whether or not the network is operating in that state.

The following is the sequence of calls for a node joining the network:

1. `emberNetworkInit(…)`
2. If not joined (`emberNetworkInit` returns `EMBER_NOT_JOINED`)...
    1. `emberSetInitialSecurityState(…)`
    2. `emberJoinNetwork(…)`

Note that if `emberNetworkInit` returned `EMBER_SUCCESS` and `emberStackStatusHandler` signaled `EMBER_NETWORK_UP`, the device will have resumed network operation with its previous security settings. `emberSetInitialSecurityState` should not be called in that case as it may cause the prior security settings to be lost, resulting in failure to properly communicate with other devices in the network.

For more information on the security bitmask settings, see the following two sections.

4.4.2 Preconfigured Link Key Node Security Configuration

The node with a preconfigured link key will setup the security bitmask as follows:

```
EmberInitialSecurityBitmask bitmask =
           (EMBER_STANDARD_SECURITY_MODE
           | EMBER_HAVE_PRECONFIGURED_KEY
           | EMBER_REQUIRE_ENCRYPTED_KEY);
```

The `EmberInitialSecurityState` should be set with a value for preconfiguredKey of the preconfigured link key.

If the trust center sends the network key in-the-clear to a device configured this way, it rejects that key and fails the join with an error code of `EMBER_RECEIVED_KEY_IN_THE_CLEAR`. This is a precaution that helps to insure that the device joins the correct network and does not communicate sensitive data to the wrong devices.

4.4.3 No Preconfigured Link Key Node Security Configuration

This is the simplest configuration for a device joining the network. It has no knowledge of any keys as the security data is sent in-the-clear. The security bitmask is as follows:

```
EmberInitialSecurityBitmask bitmask =
            (EMBER_STANDARD_SECURITY_MODE
            | EMBER_GET_LINK_KEY_WHEN_JOINING);
```

After the device receives the network key, it requests a trust center link key. If a trust center link key is not received then the join fails with the error code `EMBER_NO_LINK_KEY_RECEIVED`.

4.4.4 Application Link Keys

The decision whether or not to have application link keys is based on whether a non-trust center device will need APS encryption to send messages to another non-trust center device.

Application link keys are not required for normal operation in the network and are only used by the application. This is in contrast with trust center link keys which may be used by either the stack or the application.

4.4.5 Trust Center

The trust center has a number of decisions to make about how to setup security for the network.

1. Will devices be expected to have a preconfigured link key to join the network initially, or will the key be sent in-the-clear?
2. Will the device be forming a network without a trust center (distributed trust center mode)?
3. If the network is operating with a trust center the following decisions must be made: What type of trust center link key will be used?

   Global: Key that is the same for all devices on the network.
   Hashed: Key that is unique for all devices but is derived from a root key.
   Unique: Key that is completely unique for a particular device.
   Mixed: A combination of the three above.
4. Must ZigBee 2006 devices, which have a preconfigured network key, be supported?

The trust center must always set up a network key. It should set up a link key before forming the network, regardless of whether or not it will send the key in-the-clear or use a preconfigured one. Even if the key is sent in-the-clear, devices should obtain a link key so that they may rejoin later.

The trust center's decision to send keys in-the-clear or not is controlled by the security bitmask. See section 4.6, The Trust Center Join Handler, for more information on controlling how devices are allowed to join the network.

The following is the sequence of events for a trust center.

1. `emberNetworkInit(…)`
2. If not joined (`emberNetworkInit` returns `EMBER_NOT_JOINED`)...
   1. `emberSetInitialSecurityState(…)`
   2. `emberFormNetwork(…)`

Note that if `emberNetworkInit` returned `EMBER_SUCCESS` and `emberStackStatusHandler` signaled `EMBER_NETWORK_UP`, the device will have resumed network operation with its previous security settings. `emberSetInitialSecurityState` should not be called in that case as it may cause the prior security settings to be lost, resulting in failure to communicate properly with other devices in the network.

For more information on the security bitmasks passed to `emberSetInitialSecurityState(…)` see the following two sections.

4.4.6 Distributed Trust Center Mode Configuration

The following is the configuration for the coordinator forming a network that will operate without a trust center.

```
EmberInitialSecurityBitmask bitmask =
            (EMBER_STANDARD_SECURITY_MODE
            | EMBER_DISTRIBUTED_TRUST_CENTER_MODE
            | EMBER_GLOBAL_LINK_KEY
            | EMBER_HAVE_PRECONFIGURED_KEY
            | EMBER_HAVE_NETWORK_KEY);
```

The coordinator should set a network key and a link key so that devices on the network can obtain both. Every router that joins the network mimics some of the behaviors of the trust center. Devices that join to a router receive all their security data from that router. Each router has the choice of determining whether or not to allow a device into the network through the trust center join handler.

4.4.7 Global Trust Center Link Keys

This configuration is the simplest trust center configuration. All devices have the same trust center link key. Whether or not this key is preconfigured is determined by the trust center join handler.

```
EmberInitialSecurityBitmask bitmask =
             (EMBER_STANDARD_SECURITY_MODE
             | EMBER_GLOBAL_LINK_KEY
             | EMBER_HAVE_PRECONFIGURED_KEY
             | EMBER_HAVE_NETWORK_KEY);
```

4.4.8 Hashed Link Keys

This configuration enables the trust center to use different link keys for each device on the network, but only store one key. The preconfiguredKey in the trust center's EmberInitialSecurityState structure is the root key used to derive all the other keys. Whether or not this key is preconfigured is determined by the trust center join handler.

```
EmberInitialSecurityBitmask bitmask =
            (EMBER_STANDARD_SECURITY_MODE
            | EMBER_TRUST_CENTER_USES_HASHED_LINK_KEY
            | EMBER_HAVE_PRECONFIGURED_KEY
            | EMBER_HAVE_NETWORK_KEY);
```

4.4.9 Unique Link Keys with a Global Trust Center Link Key

This configuration enables the trust center to have a unique link key with one or more devices, and a global trust center link key for all other devices. This requires that the trust center have the link key library compiled into the image.

The following is the security bitmask for this configuration:

```
EmberInitialSecurityBitmask bitmask =
            (EMBER_STANDARD_SECURITY_MODE
            | EMBER_GLOBAL_LINK_KEY
            | EMBER_HAVE_PRECONFIGURED_KEY
            | EMBER_HAVE_NETWORK_KEY);
```

The trust center should perform the following additional steps to make this configuration work:

1. For each joining device for which the trust center wants to configure a link key, call Link Key Table API to add a link key, such as:

   `emberAddOrUpdateKeyTableEntry(address, TRUE, keyDataPointer)`
2. Join the device(s) to the network.

If the device joins before the unique trust center link key is set up then the trust center will try to use the global or hashed link key.

4.4.10 Security Setup for ZigBee Home Automation (ZHA) Devices

The ZHA application profile is based on ZigBee standard security. Therefore it uses a shared 128-bit network key for encryption of all traffic at the NWK layer and above. This key is typically chosen at random upon network formation.

Additionally, ZHA uses a global, well-known trust center link key so that any HA-certified devices can potentially join the network (if it permits new devices) with this preconfigured key rather than requiring a network-specific configuration. This makes security configuration and joining/authentication much easier for ZHA devices. However, it also makes ZHA networks only mildly secure, since any device with knowledge of this well-known link key can appear “authorized” to the trust center and request a delivery of the network key (through the standard NWK layer rejoin mechanism).

In ZHA networks, due to the use of a well-known, global trust center link key, the application designer may want to consider adding mechanisms to provide more control over which devices enter the network (see section 4.6, The Trust Center Join Handler, later in this document) or to allow the user an interface by which to reject unwanted devices (through the ZDO Leave Request) or let the device remove itself from the network (by calling `emberLeaveNetwork`) after they have erroneously entered the network.

Devices joining ZHA networks use the following initial security bitmask:

```
EmberInitialSecurityBitmask bitmask =
            (EMBER_STANDARD_SECURITY_MODE
           | EMBER_HAVE_PRECONFIGURED_KEY
           | EMBER_GLOBAL_LINK_KEY
           | EMBER_REQUIRE_ENCRYPTED_KEY);
```

The preconfigured global link key is set according to the value defined in the ZigBee Home Automation Public Application Profile specification (ZigBee document 053520). Refer to that document (found at the http://www.zigbee.org website) for this and more information concerning the ZHA profile.

4.4.11 Security Setup for ZigBee Smart Energy (ZSE) Devices

The ZSE application profile is based on ZigBee standard security but with a few additional requirements at the application layer. In addition to using a shared network key chosen at random during network formation, ZSE networks use unique, device-specific trust center link keys to secure APS-layer communications. Preconfigured link keys are generated through the use of an AES MMO hash algorithm over a variable-length installation code preprogrammed during the manufacturing process. ZSE networks also employ Certificate-Based Key Establishment (CBKE) and Elliptic Curve Cryptography (ECC) to authorize devices and create secure, random trust center link keys to use for ZSE communication.

Devices joining ZSE networks generally use the following initial security bitmask for deployed networks:

```
EmberInitialSecurityBitmask bitmask =
            ( EMBER_STANDARD_SECURITY_MODE
            | EMBER_GET_PRECONFIGURED_KEY_FROM_INSTALL_CODE
            | EMBER_REQUIRE_ENCRYPTED_KEY);
```

Alternatively, some ZSE devices may use the following configuration during development/testing operation, where a global, preconfigured trust center link key is desirable for ease of debugging and setup. Please note that this is NOT a certifiable configuration.

```
EmberInitialSecurityBitmask bitmask =
            ( EMBER_STANDARD_SECURITY_MODE
            | EMBER_HAVE_PRECONFIGURED_KEY
            | EMBER_GLOBAL_LINK_KEY
            | EMBER_REQUIRE_ENCRYPTED_KEY);
```

For more information regarding ZSE security, refer to section 5, ZigBee Smart Energy (ZSE) Security.

4.5 Error Codes Specific to Security

A number of error codes pertaining to security may be returned by the stack. Table 3 describes the error codes and when they may be received.

Table 3. Error Codes Specific to Security

Error Code	Description
EMBER_SECURITY_STATE_NOT_SET	The device did not successfully call emberSetInitialSecurityState(…) to set the initial security parameters before forming or joining the a secure ZigBee PRO network.
EMBER_NO_NETWORK_KEY_RECEIVED	The device failed to join a secured ZigBee PRO network because it did not receive the network key sent from the trust center. This may also occur when operating in distributed trust center mode.
EMBER_NO_LINK_KEY_RECEIVED	The device failed to join the network because it did not receive a response to its request for a link key. The device did receive the network key but failed the join because it specified that it wanted a link key (EMBER_GET_LINK_KEY_WHEN_JOINING). This may also occur in distributed trust center mode.
EMBER_RECEIVED_KEY_IN_THE_CLEAR	The device failed to join because it specified that it had a preconfigured key and required that the network key must be sent encrypted using the preconfigured link key (EMBER_REQUIRE_ENCRYPTED_KEY), but the trust center sent the key in the clear (EMBER_SEND_KEY_IN_THE_CLEAR). This may also occur in distributed trust center mode.
EMBER_PRECONFIGURED_KEY_REQUIRED	The device failed to join because it did not specify a preconfigured key and the trust center sent the network key encrypted using a preconfigured key (EMBER_USE_PRECONFIGURED_KEY). This may also occur in distributed trust center mode.
EMBER_APS_ENCRYPTION_ERROR	The application requested APS encryption for a message but the stack was unable to encrypt the message. This could be because the long address corresponding to the short address of the destination is not known, or no link key exists between the destination and local node.
EMBER_KEY_INVALID	The passed key data is not valid. A key of all zeros or all F's is a reserved value and cannot be used.
EMBER_INVALID_SECURITY_LEVEL	The chosen security level (the value of EMBER_SECURITY_LEVEL) is not supported by the stack.
EMBER_KEY_TABLE_INVALID_ADDRESS	There was an attempt to set an entry in the key table using an invalid long address. An entry cannot be set using either the local device's or trust center's IEEE address. Or an entry already exists in the table with the same IEEE address. An address of all zeros or all F's is not a valid address in 802.15.4.
EMBER_TOO_SOON_FOR_SWITCH_KEY	There was an attempt to broadcast a key switch too quickly after broadcasting the next network key. The trust center must wait at least a period equal to the broadcast timeout so that all routers have a chance to receive the broadcast of the new network key.
EMBER_SIGNATURE_VERIFY_FAILURE	
EMBER_KEY_NOT_AUTHORIZED	The message could not be sent because the link key corresponding to the destination is not authorized for use in APS data messages. APS commands (sent by the stack) are allowed, but to use it for encryption of APS data messages, it must be authorized using a key agreement protocol (such as CBKE).

4.6 The Trust Center Join Handler

The decision whether or not to allow devices onto the network is separate from the initial security configuration. This allows the trust center the flexibility to take other criteria into consideration (for example, button presses or IEEE addresses) when determining what to do. The callback is:

```
EmberJoinDecision
emberTrustCenterJoinHandler(EmberNodeId newNodeId,
                                          EmberEUI64 newNodeEui64,
                                          EmberDeviceUpdate status,
                                          EmberNodeId parentOfNewNode);
```

The trust center join handler is called whenever a device joins or rejoins the network. It informs the trust center application about the node, including what operation the device is performing.

There are several possible status codes:

- The device left (EMBER_DEVICE_LEFT)
- A secured rejoin was performed (EMBER_STANDARD_SECURITY_SECURED_REJOIN)
- An unsecured join was performed (EMBER_STANDARD_SECURITY_UNSECURED_JOIN).
- An unsecured rejoin was performed (EMBER_STANDARD_SECURITY_UNSECURED_REJOIN).

For the first two cases (a leave or secure rejoin), no action is required by the trust center (EMBER_NO_ACTION). These cases are purely informative and the trust center can decide what (if anything) to do with the information.

For the third and fourth cases, the specification does differentiate between an unsecured join and unsecured rejoin, but the following should be noted: a malicious device could try either one to gain access to the network, so the trust center should treat them both carefully.

If the trust center is not allowing new devices on the network and does not believe this device was previously part of the network, it may simply deny the join (EMBER_DENY_JOIN).

If the trust center is in a state where it is not accepting any new devices, it may assume that this is a rejoin. For a rejoin, the trust center should send back the network key encrypted with the device's trust center link key (EMBER_USE_PRECONFIGURED_KEY).

If the trust center is allowing new devices on the network, it must choose whether or not to send the network key in the clear. If the trust center expects the device should have a preconfigured link key, it may send the network key encrypted (EMBER_USE_PRECONFIGURED_KEY). If the trust center allows devices to join without any preconfigured key, it may send the network key in the clear (EMBER_SEND_KEY_IN_THE_CLEAR).

A default implementation has been provided for the `emberTrustCenterJoinHandler(…)`. A global Boolean associated with that default handler, `emberDefaultTrustCenterJoinDecision`, controls whether that default handler sends the key in-the-clear or requires a preconfigured link key. By default, the default implementation of the trust center join handler sends the network key encrypted using the link key (EMBER_USE_PRECONFIGURED_KEY).

4.6.1 EZSP Security Policy

On EZSP-based NCP platforms, the application will not be asked directly to make a decision about whether or not the device can join the network. Instead the host application can set a security policy on the NCP that determines globally whether or not devices may join the network, and whether they are expected to have a preconfigured link key. The policies are shown in Table 4.

Table 4. EZSP Security Policies

Policy	Description
EZSP_ALLOW_JOINS	Allows new devices that do not have a preconfigured key to join. The network key is sent in-the-clear.
EZSP_ALLOW_PRECONFIGURED_KEY_JOINS	Allows new devices that have a preconfigured key to join the network. The network key is sent APS-encrypted with the appropriate trust center link key.
EZSP_ALLOW_REJOINS_ONLY	Only allows devices to rejoin the network. Secure rejoins do not require any action by the trust center, but it is notified that this event has occurred. Unsecure rejoins cause the trust center to send the network key APS-encrypted with the appropriate trust center link key. This is the default.
EZSP_DISALLOW_ALL_JOINS_AND_REJOINS	All attempts to join or rejoin the network are denied. The trust center sends back a message to the parent of the (re)joining device to remove the device from the network.

If the host application decides to set a policy that allows joins, it should do so for a limited time. After all the devices have joined the network, the host should change the NCP's policy to EZSP_ALLOW_REJOINS_ONLY. This ensures that only devices that are a part of the network and have a link key can successfully rejoin.

The NCP firmware initiates a callback up to the host application to inform it that a security event has occurred and the decision that was made based on the current security policy.

```
void ezspTrustCenterJoinHandler(EmberNodeId newNodeId,
                                EmberEUI64 newNodeEui64,
                                EmberDeviceUpdate status,
                                EmberJoinDecision policyDecision);
```

4.7 Security Settings After Joining

The security settings for a device cannot be changed after it has formed or joined the network. Information may be obtained regarding the security settings the device is currently using, and the values of the keys.

The security settings may be obtained with the following call:

```
EmberStatus emberGetCurrentSecurityState(EmberCurrentSecurityState *state)
```

4.7.1 Current Security State Structure

```
typedef struct {
  EmberCurrentSecurityBitmask bitmask;
  EmberEUI64 trustCenterLongAddress;
} EmberCurrentSecurityState;
```

4.7.2 Current Security Bitmask

The bitmask of the current security settings is shown in Table 5.

Table 5. Bitmask of Current Security Settings

Bits	Name	Information Applies to	Description
1	EMBER_DISTRIBUTED_TRUST_CENTER_MODE	All devices	Specifies whether the device is currently operating in a network with (0) or without (1) a trust center. If set to zero, the trustCenterLongAddress field does not contain any valid data.
2	EMBER_GLOBAL_LINK_KEY	Trust center only	Specifies whether the trust center is using the same link key for all devices (1) or separate link keys for each device (0).
3	Reserved		
4	EMBER_HAVE_TRUST_CENTER_LINK_KEY	Non trust center devices	Specifies whether the device has a trust center link key (1) or if no link key is available (0).
5–6	Reserved		
2,7	EMBER_TRUST_CENTER_USES_HASHED_LINK_KEY	Trust center	If set (1), indicates that the trust center is using a hashed link key to derive individual link keys from the preconfigured root key.
8-15	Reserved		

4.7.3 Obtaining the Security Keys

The current security keys may be obtained with this API call:

```
boolean emberGetKey(EmberKeyType type, EmberKeyStruct* keyStruct)
```

The call fetches the requested key type and copies the data into the passed keyStruct parameter.

There are several security key types:

- Trust center link key (EMBER_TRUST_CENTER_LINK_KEY)
- Application link key (EMBER_APPLICATION_LINK_KEY)
- Current network key (EMBER_CURRENT_NETWORK_KEY)
- Next network key (EMBER_NEXT_NETWORK_KEY)

Note: Trust center master keys and application master keys are not used in ZigBee standard security and therefore are not supported in EmberZNet PRO. These enumerations exist as placeholders for possible future functionality.

- Trust center master Key (EMBER_TRUST_CENTER_MASTER_KEY)
- Application master key (EMBER_APPLICATION_MASTER_KEY)

4.7.4 The Key Data Structure

The EmberKeyStruct contains key data as well the associated information about the key. The fields in the key structure contain valid information based on the key structure bitmask.

```
typedef struct {
  EmberKeyStructBitmask bitmask;
  EmberKeyType type;
  EmberKeyData key;
  int32u outgoingFrameCounter;
  int32u incomingFrameCounter;
  int8u sequenceNumber;
  EmberEUI64 partnerEUI64;
} EmberKeyStruct;
```

4.7.5 The Key Structure Bitmask

Table 6 describes the key structure bitmask.

Table 6. Key Structure Bitmask

Bits	Name	Description
0	EMBER_KEY_HAS_SEQUENCE_ NUMBER	When set (1), indicates a valid sequence number in the sequenceNumber field of the EmberKeyStruct
1	EMBER_KEY_HAS_OUTGOING_ FRAME_COUNTER	When set (1), indicates a valid frame counter in the outgoingFrameCounter field of the EmberKeyStruct
2	EMBER_KEY_HAS_INCOMING_ FRAME_COUNTER	When set (1), indicates a valid frame counter in the incomingFrameCounter field of the EmberKeyStruct
3	EMBER_KEY_HAS_PARTNER_EUI64	When set (1), indicates a valid EUI64 Address in the partnerEUI64 field of the EmberKeyStruct
4-15	Reserved	-

4.8 Link Keys Library

Normally a device can only store one link key, the trust center link key. However, those devices that wish to store more than one link key (needed for application link keys) have the link keys library.

1. The link keys library can be used in one of two ways: A trust center that wishes to store unique link keys (not hashed or global) for specific devices on the network and keep track of the incoming frame counters for those devices.
2. A non-trust center device that wishes to use application link keys to talk with one or more non-trust center device on the network.

The link key library uses a table stored in flash and RAM to keep track of the keys and their associated data. The table's size is configurable through the EMBER_KEY_TABLE_SIZE definition. By default the table's size is 0.

Each entry in the table has the following elements stored in flash:

- The key data (16-bytes)
- The EUI64 associated with the partner that shares the key (8-bytes).
- Information about the key (1-byte).

Each entry in the table also stores the following elements in RAM:

- The partner's incoming APS frame counter (4-bytes)

The application should take these requirements into consideration when sizing the key table.

Normal nodes use the library for different reasons and thus each device type (trust centers and non trust centers) can decide independently whether to use the library.

4.8.1 Trust Center Using the Link Keys Library

The trust center can use the link keys library to store unique trust center link keys for specific devices on the network. Using the a unique trust center link key for one or more devices does not prohibit the trust center from also using global or hashed trust center link keys for all other devices on the network. In fact, unless a finite number of devices is on the network and the trust center has enough flash and RAM to store unique link keys for every device, a global or hashed trust center link key is recommended to supplement the limited storage of the link key table.

For a device that has a unique trust center link key the following procedure should be followed. Before that device joins the network, the trust center should set up the link key in the table by calling into the Link Key Table API. The joining device does not need to use the link keys library since the core security library has storage space for the trust center link key. The joining device can either preconfigure that key when calling `emberSetInitialSecurityState(…)`, or it can simply request a link key from the trust center. In both cases, the trust center uses the link key set up in the table for that device.

When using a mix of hashed or global link keys and unique link key entries in the key table, the trust center always consults the key table first to find a specific entry for a device. If a specific entry in the key table does not exist, then it falls back on the global or hashed method to determine the link key for that device.

The trust center does not need to include the link keys library to answer requests for application link keys. The code to process the request is in the core security library.

4.8.2 Normal Nodes using the Link Keys Library

Normal nodes (non-trust center devices) need the link keys library only if they wish to use application link keys to encrypt messages to other devices on the network. An application can obtain an application link key in one of two ways:

1. Manual configuration by adding the link key to the table.
2. Asking the trust center for one.

Two nodes can manually setup an application link key between themselves by each agreeing on a key and then calling into the Link Key Library API to set a specific entry in the table. Once the key is in place the devices can communicate using APS encryption.

Alternatively both nodes may contact the trust center to obtain an application link key. The advantage of contacting the trust center is that both devices are using their trust center link keys to securely request and receive an application link key. Both devices use the trust center to establish a trust relationship with one another.

After sending a request for an application link key, sleepy and mobile devices should poll at a higher rate until the key is successfully established or the operation times out. The ultimate result of the attempt to establish a link key is returned by the `emberKeyEstablishmentHandler(…)`.

The length of time a requesting device waits is configurable by the EMBER_REQUEST_KEY_TIMEOUT configuration item. The default value is 2 minutes, but it can be set to a value between 1 and 10 minutes. This controls how long a normal node waits before returning a value of EMBER_KEY_ESTABLISHMENT_TIMEOUT through the `emberRequestKeyEstablishmentHandler(…)`.

On the trust center this configuration value controls how long the trust center waits for a pair of matching key requests. The `EMBER_REQUEST_KEY_TIMEOUT` value should be set the same on both the trust center and all other devices in the network.

4.8.3 Link Key Table API

When the trust center sends new keys to the devices, the stack automatically updates the table and notifies the application. Otherwise it is up to the application to manage the link key table. The stack never deletes an entry from the table.

Table 7 contains the API calls pertaining to the link keys library.

Table 7. Link Keys Library API Calls

Link Key Table Function	Description
`boolean emberSetKeyTableEntry(int8u index, EmberEUI64 address, boolean linkKey, EmberKeyData* keyData)`	Sets the key, address, and type in the table to the specified data. If the linkKey parameter is false then a master key is implied. The incoming frame counters associated with that key are reset to 0. If the index is invalid or the address or key is all 0s or all Fs, the operation fails and FALSE is returned.
`boolean emberGetKeyTableEntry(int8u index, EmberKeyStruct* result)`	Retrieves the key at the specified entry. If the entry is empty or the index is out of range, then FALSE is returned and the result data structure is not populated.
`boolean emberAddOrUpdateKeyTableEntry(EmberEUI64 address, boolean linkKey, EmberKeyData* keyData)`	First attempts to update an existing key entry matching the passed address. If no entry matches the address, then it searches for an empty entry. If successful it sets the address, key type (link or master) and key data for the new entry. The incoming frame counter for that key is set to 0. If no existing entry and no free entry can be found, then the operation fails and the function returns FALSE.
`int8u emberFindKeyTableEntry(EmberEUI64 address, boolean linkKey)`	Searches for the key entry matching the passed address and key type. If a matching entry is found then the index is returned. Otherwise 0xFF is returned. To search for an empty key table entry pass in an address of all zeroes.
`boolean emberEraseKeyTableEntry(int8u index)`	Erases the data in the key table at the specified index. If the index is out of range then the operation fails and FALSE is returned.
`EmberStatus emberRequestLinkKey(EmberEUI64 partner)`	(Valid only for non trust center devices) Sends a request to the trust center for a key with the associated partner device. If the partner address is the trust center, then the request is for a trust center link key. A key is immediately sent back to the device. If the partner address is not the trust center, then the trust center stores the request until the partner device also requests a key. Then an application link key is randomly generated and sent back to both devices. The function returns EMBER_SUCCESS if the request was sent. The success or failure of the operation is returned by the emberKeyEstablishmentHandler(). If a key already exists, then on successful receipt of the new key the old key isreplaced.
`void emberKeyEstablishmentHandler(EmberKeyStatus status)`	A callback to the application about an attempt to establish a key. The result is returned in the passed parameter. It is not called during joining if EMBER_GET_LINK_KEY_WHEN_JOINING is set and the device gets a trust center link key.

The following is the EmberKeyStatus data structure that defines the result of an attempt to establish a key.

```
enum EmberKeyStatus
{
         EMBER_APP_LINK_KEY_ESTABLISHED = 0,
         EMBER_APP_MASTER_KEY_ESTABLISHED = 1,
         EMBER_TRUST_CENTER_LINK_KEY_ESTABLISHED = 2,
         EMBER_KEY_ESTABLISHMENT_TIMEOUT = 3,
         EMBER_KEY_TABLE_FULL = 4,
};
```

Note: Although the link key table keeps track of whether the key is a master key or link key, only the link key type is used by the stack. EmberZNet PRO does not support using master keys to derive link keys.

4.9 APS Encryption

Applications may add APS-level encryption to their messages as an extra layer of security. Network layer encryption is always used for application messages sent in a secured network. However APS encryption on top of network layer encryption offers several advantages:

1. Only the sending and receiving devices should have the link key used for encrypting and decrypting messages. An exception is messages sent to or from a trust center, where the trust center is using a global trust center link key.
2. Additional protection against replay attacks is present for APS-encrypted messages.

APS encryption requires that a pair of devices wanting to use it establish a link key. If one of the devices is the trust center then this is already set up through the trust center link key established when joining the network. If neither device is the trust center, then the link keys library must be used and an application link key must be set up through that API.

Once a link key is established, the devices can apply APS-layer encryption by setting a bit in the EmberApsStruct that indicates APS encryption is required for sending this message. If a message is received that has APS encryption then the bit is set accordingly.

```
enum EmberApsOption
{
  ...
  /** Send the message using APS Encryption, using the link key shared
      with the destination node to encrypt the data at the APS Level. */
  EMBER_APS_OPTION_ENCRYPTION              = 0x0020,
  ...
```

Application messages that have APS encryption are decrypted automatically and passed up to the application. It is up to the application to decide whether or not to accept or reject them. Certain ZigBee messages (APS commands) generated by the stack require APS encryption and are rejected silently if they are not encrypted.

APS encryption requires additional overhead and consumes more of the message payload. APS encryption uses 9 bytes of the payload (5 bytes for a security reader and a 4-byte MIC).

4.9.1 The Short to Long Address Mapping

In order to properly decrypt an APS-encrypted message, the receiving device must know the long address of the sending device. The long address of the sender is not present in the APS-encrypted message; therefore it is advised that the application store the short-to-long address mapping of all devices from which it wishes to send/receive APS-encrypted messages.

If a device cannot determine the long address from the short address of an APS-encrypted message, encryption fails and the message is silently discarded.

The short to long address map may be stored in one of several places:

- Neighbor table
- Child table
- Security tokens (trust center address only)
- Address table (including the Security Address Cache)
- Binding table

The neighbor and child tables are managed by the stack and will maintain the address mapping of those devices as long as they continue to be children or neighbors of the local device. Since it is possible for both neighbors and children to come and go, it is advised not to count on this automatic mapping. Instead, the application should add an entry in the address table or binding table for its own use. To facilitate address resolution during APS security operations as part of authenticating joining/rejoining devices, applications that can function as the trust center may use the security address cache (a subset of the address table that can be optionally managed by the Ember security utilities module).

The trust center's long address is stored in a joining device's tokens. The short address is always that of the coordinator, 0x0000.

4.10 Updating and Switching the Network Key

Changing the network key is a two-step process in which the trust center first broadcasts a copy of the next network key, and then tells devices to switch to using the next network key.

Note: Updating and switching the network key is not available when running in distributed trust center mode.

To update the network key, this call is made:

```
EmberStatus emberBroadcastNextNetworkKey(EmberKeyData* key)
```

This call broadcasts the next network key throughout the network with the next key sequence number. The message is encrypted using the current network key. Once a new network key has been updated, it must be used. Attempts to broadcast a different network key before switching fail.

At a minimum, the trust center should wait a period equal to the broadcast timeout before switching to the new network key. This insures that all routers in the network have received the next network key.

To switch to the next network key, make this call on the trust center:

```
EmberStatus emberBroadcastNetworkKeySwitch(void)
```

The command to switch to the new network key causes all devices that hear the message (and have the next network key) to start encrypting all outgoing messages with the next network key. Their outgoing network frame counters are reset to zero.

Note: It is important that the last network key is still available to the node and can be used to decrypt incoming network messages. To completely deprecate a key from being used, the network key must be updated and switched twice.

4.10.1 Sleepy and Mobile End Devices

Sleepy and mobile end devices may miss a network key update if they did not poll their parent in time to hear the broadcast of the next network key. Ember routers automatically keep track of sleepy and mobile devices that miss a

network key update. If the end device polls after missing a key update and switch, the router informs the child that it needs to perform an unsecured rejoin.

End devices automatically perform an unsecured rejoin upon receiving that message. Because it is possible that the rejoin may fail due to any number of networking issues (such as a bad link), the application may need to make a call to rejoin the network on its own.

4.10.2 Notification of a Switch to a New Network Key

All devices can be notified if there is an update and switch to a new network key using this application-defined callback:

```
void emberSwitchNetworkKeyHandler(int8u sequenceNumber);
```

This callback is made by the stack when it changes the current network key it is using. The sequence number of the new key is passed back. Information about the new key in use may be retrieved by a call to `emberGetKey(…)`.

4.11 Rejoining the Network

A device may need to rejoin the network if it has lost connectivity with its parent or has missed a network key update. To rejoin the network, make this API call:

```
EmberStatus emberRejoinNetwork(boolean hasCurrentNetworkKey)
```

If the device believes it has simply lost its parent, it may set the `hasCurrentNetworkKey` parameter to TRUE, which causes it to perform a secure rejoin. This is the quickest way to get back on the network and the device need only receive a response from its new parent in order to start communicating again.

If the device believes it missed a key update, or an attempt to rejoin with `hasCurrentNetworkKey` of TRUE has failed, the device should attempt an unsecure rejoin by setting the `hasCurrentNetworkKey` parameter to FALSE. This method takes slightly longer to get back on the network, because the trust center must be consulted and an updated network key must be issued to the rejoining device.

The `hasCurrentNetworkKey` parameter is ignored when operating in a network without security.

4.12 Transitioning From Distributed Trust Center Mode to Trust Center Mode

When a device wishes to change a network operating without a trust center to one operating with a trust center (and become the trust center), these requirements must be met:

- The device must have the address of the coordinator (0x0000).
- It must have the current network key and global preconfigured link key.
- It must be joined into the network.

If all those criteria are met, it may make this call:

```
EmberStatus emberBecomeTrustCenter(EmberKeyData* newKey)
```

This call causes the device to broadcast the next network key and indicate to all devices that it is the new trust center. Devices immediately switch to trust center mode at this point, but they do not switch to the new network key.

The new trust center should also switch the network to the next network key by calling `emberBroadcastNetworkKeySwitch(void)`. It should follow guidelines for updating the network key outlined in section 4.8, Link Keys Library.

The new trust center must also take into consideration the requirements presented in section 3.7, Additional Requirements for a Trust Center.

5 ZigBee Smart Energy (ZSE) Security

Due to the complexity of the security used in ZSE application development, this topic has been given its own section detailing the implementation and design requirements and considerations. Note that more detailed information about ZSE can be found in the ZigBee Smart Energy Profile Specification, available at the http://www.zigbee.org website.

5.1 Overview

The following sections explain the concepts used in ZSE to provide additional security beyond the ZigBee standard security.

5.1.1 Installation Codes

ZSE devices use unique, preconfigured trust center link keys to enter the ZSE network and obtain the current network key. However, since this unique key must be known to both the joining device and the trust center at the time of network entry, a piece of shareable data known as the “installation code” (sometimes also referred to as the “install code”) is used to derive the key at both sides. The code can be any arbitrary value of 6, 8, 12 or 16 bytes, followed by a 16-bit CRC (least significant byte first) over those bytes.

This code is then used as the input to a Matyas-Meyer-Oseas (MMO) hash function (as specified in ZigBee Document 053474, the ZigBee Specification), with a digest size (hash length) equal to 128 bits. The 128-bit (16-byte) result of this AES-MMO hash function is used as the value for the preconfigured trust center link key for that device, and the trust center can then install a key table entry with that key and the EUI64 of the joining device, which then allows the authentication to take place successfully during joining, and the joining device can successfully receive and decrypt the network key delivery. See section 4.4.11, Security Setup for ZigBee Smart Energy (ZSE) Devices, for the initial security bitmask generally used by devices joining ZSA networks.

As part of this process, the installation code and the joining device’s EUI64 must be conveyed out-of-band (outside of the target ZigBee network, since the new node is not yet joined) to the network’s trust center to allow the proper link key table entry to be created. This communication could involve the device installer contacting the ZSE network administrator (the party responsible for the trust center, such as a utility company) by telephone or Internet to provide the necessary information. More information about this process can be found in the “Out of Band PreConfigured Link Key Process” section of the ZigBee Smart Energy Profile Specification.

5.1.2 Certificates and Key Establishment

Once it has joined to the ZSE network and obtained the random, shared network key from the trust center, the new device must then re-negotiate its trust center link key through certificate-based key establishment (CBKE). This key negotiation protocol with the trust center ensures that the new link key is unrelated to the preconfigured key, ensures a key that is random and irreproducible, and provides proof of identity by validating the authenticity of the certificates at both devices. The new link key derived from this CBKE process replaces the original, preconfigured trust center link key, such that the preconfigured key is not used again unless this new ZSE device is purged from the network and later needs to re-enter. ZSE networks require that a CBKE-based link key shall be used for unicast data communications on most ZSE clusters. (Refer to the “Cluster Usage of Security Keys” section of the ZigBee Smart Energy Profile Specification for details about which clusters require only Network layer security and which require both Network and APS layer security.

CBKE is variation of Public Key-Key Establishment (PKKE, as opposed to SKKE, Symmetric Key-Key Establishment) between a pair of devices. PKKE is a process whereby a link key is established based on each party’s shared, static, public key and ephemeral, public key. Since these keys are public, they do not require secrecy in their storage and transmission. These keys by themselves (without the non-public certificate data) aren’t enough to recreate the key, so knowledge of these public keys doesn’t compromise the established link key. In CBKE, specifically, each device’s static, public key is transported as part of a device-implicit certificate signed by

the sender's certificate authority (CA), allowing the receiver to validate the device's identity during key establishment; this differs from traditional PKKE, where certificates are manually created.

The digital certificates used in the CBKE process are programmed into each device at manufacturing time and are issued by the CA. For the process to complete successfully, both devices must contain certificates signed by the same CA. For ZSE networks using Smart Energy 1.x protocol versions, Certicom (www.certicom.com) is the only ZigBee-approved certificate issuer. Certicom offers certificates signed by either of the following CAs:

- Test SE CA – A special certificate authority used exclusively for non-commercial testing purposes. Certificates signed by this CA are free to generate through Certicom's website.
- Production CA – The normal certificate authority used by Certicom to sign certificates for production-grade devices used in commercial deployments. These certificates require paid licensing terms with Certicom to generate and will not interoperate with test certificates signed by the Test SE CA.

The certificate data stored on each device consist of the fields described in Table 8.

Table 8. ZSE Stored Certificate Fields

Size (bytes)	Name	Description
22	CA Public Key	Public key specific to the CA who signed the certificate. During CBKE, this is used to verify the authenticity of the CA.
48	Device-Implicit Certificate	Unique data signed by the CA (using the CA's private key) and representing the digital certificate for this specific device. This is the portion of the certificate that is shared over the air during CBKE and contains the following subfields: • Reconstruction data for the device's public key (22 bytes) • This device's IEEE MAC address (EUI64), also known as the Subject for the certificate (8 bytes in MSB order) • Issuer ID for the CA who created this device-implicit certificate (8 bytes in MSB order) • Profile-specific data as defined by the ZigBee application profile using the certificate. The first 2 bytes represent the 16-bit ZigBee application profile ID (in most significant byte notation), such as 0x0109 for ZSE. (10 bytes)
21	Device Private Key	A unique, device-specific value chosen by the CA during certificate generation. During CBKE, this is used as an input (along with the Device Public Key) to an Elliptic-Curve Cryptography (ECC) algorithm.

This certificate data is used at runtime to establish a shared secret (the new link key) through ECC computations along an elliptic-curve. While the computational details of the CBKE process are beyond the scope of this document, the ZigBee Cluster Library (ZCL) messages exchanged as part of this process are illustrated in Figure 12. Additional details about this process can be found in Appendix C of the ZigBee Smart Energy Profile Specification. All of these messages are encrypted at the Network layer without any APS layer encryption. The Initiator in this process is typically the new device that entered the network, while the Responder is typically the trust center or Energy Services Interface (ESI) for the Home Area Network (HAN).

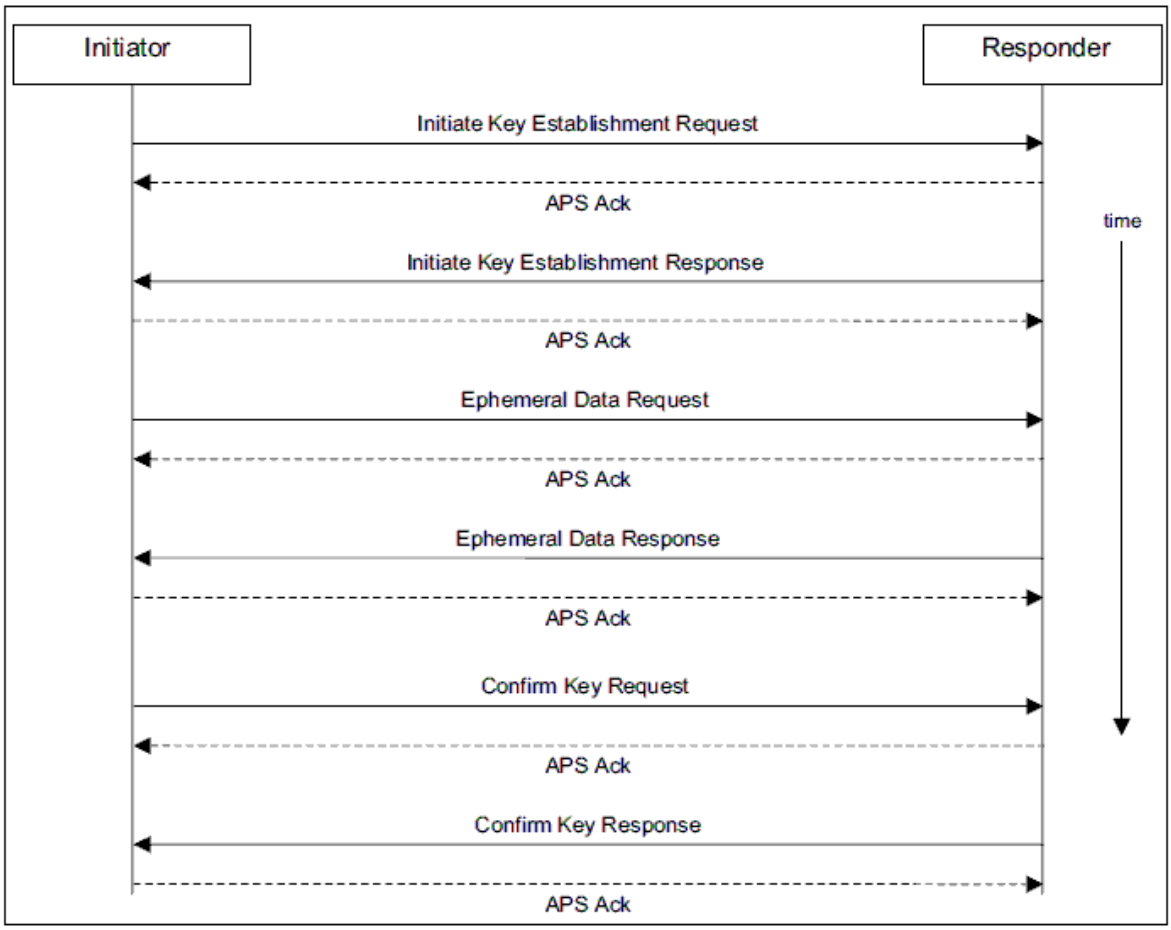

Figure 12. Message Exchange for Certificate-Based Key Establishment

Once the key has been established, it can be used for future ZSE-related transactions among this pair of devices. If ZSE-related communications is desired between another pair of devices in the HAN, the two devices can request a mutual link key by requesting one from the trust center (who is a trusted party by virtue of CBKE having succeeded). For information about third-party application link key requests, refer to section 3.2.3, Application Link Keys.

5.1.3 Application layer requirements

To ensure ZSE end-product compliance, ZSE devices have a number of special design requirements including the following, which make them unique from other ZigBee devices:

- Support for CBKE, including the underlying ECC algorithm used for key generation.
- Support for Elliptic-Curve Digital Signature Authentication (ECDSA), which is needed for validating firmware image data transferred in the ZSE Over-the-Air (OTA) Bootload cluster.
- Pre-installed certificates issued by the proper certificate authority.
- Pre-installed installation codes chosen by the manufacturer.
- Access to the installation code and EUI64 during network setup (to facilitate out-of-band communication of this data to the trust center).
- (Trust center only) Key table space sufficient to track the maximum number of ZSE devices that the HAN will support.

- (Trust center only) Support for inputting installation codes (for deriving link key values through an AES-MMO hash function) or link key values and EUI64 addresses (for creating key table entries for new devices).
- APS data messages for certain clusters (most ZSE clusters) require APS security with an application (or trust center) link key.

The Ember application framework can ensure that the above requirements are satisfied (where applicable) when the Ember AppBuilder tool is used to configure a ZSE device. For more information about Ember application framework-based development, refer to document UG102, *Application Framework Developer Guide*.

5.2 Additional Sources of Information

For more information regarding ZSE security concepts, refer to the following resources:

- *ZSE AMI Profile Specification* – ZigBee document #075356. This is the top-level application profile specification for ZSE and is available for public (non-member) download through the www.zigbee.org website. However, as of this writing (March, 2011), this document has not yet been updated to reflect the latest significant ZSE revision, Smart Energy 1.1, so some information may be missing or out of date. Beginning in April, 2011, certification on the legacy Smart Energy 1.0 standard will no longer be permitted, so developers are encouraged to utilize Smart Energy 1.1, which is implemented in the Ember application framework code provided with EmberZNet PRO releases 4.3.0 and later.
- *Smart Energy 1.1 v0.9 Draft Specification* – ZigBee document #105638. This is the current working copy of the Smart Energy 1.1 specification, which was recently approved by the ZigBee Alliance as a replacement for the existing Smart Energy 1.0 specification, but which has not been made public (to non-members) as of this writing. This document requires Participant or Promoter level membership in the ZigBee Alliance to access and is available through the ZigBee Membership Web Portal. (It is not accessible to Adopter level members.)
- Document AN708, *Setting Manufacturing Certificates and Installation Code*. This is an application note included in EmberZNet PRO releases supporting ZSE development (also available online through the www.silabs.com/zigbee-docs website). It explains how to program the necessary manufacturing data (certificates and installation codes) into Ember chips using Ember programming tools and how to verify these data once they have been programmed into the device.
- Document AN714, *Smart Energy ECC-Enabled Device Setup Process*. This application note, included in EmberZNet PRO releases, describes how to set up a device with the security resources required to support Smart Energy (SE) security. While these security resources are not necessary for testing SE networks, any devices designed to participate in or host a ZigBee-compliant, production-grade (non-test) SE network must implement these features.
- Document UG102, *Application Framework Developer Guide*. This document, included in EmberZNet PRO software releases and available online through the www.silabs.com/zigbee-docs website, describes the command line interface (CLI) commands available for inspecting and altering the security configuration of an Ember application framework-based application at runtime. It also explains any callbacks or plugins used by Ember application framework to implement security behavior in the application.
- Document UG112, *Ember AppBuilder User's Guide*. This document, included in the Ember Desktop software installation package, describes the operation and interface usage of the Ember AppBuilder software, which is a graphical front-end used to configure the Ember application framework. The document mirrors the contents of the application on-line help. It contains a chapter on security, explaining how to set up different security models in an Ember application framework-based application.

CONTACT INFORMATION

Silicon Laboratories Inc.

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

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