ember

EM250

EM260

EM35x

EM35x EZSP

Ember Application Development Fundamentals

30 March 2012

120-3029-000B

New in this Revision
Clarified language in 7.3.6. around launching the standalone bootloader in the EM35x.

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

Contents

1 Introduction

1.1 Purpose

Ember Application Development Fundamentals covers information on a number of topics that project managers, application designers and developers should understand before beginning to work on an embedded networking solution using Ember chips, the EmberZNet PRO stack, and Ember tools.

1.2 Topics

This guide covers the following information:

- **Fundamentals of Wireless Networking**: Introduces some fundamental concepts of wireless networking. These concepts are referred to in other chapters of this guide. If you are new to wireless networking, you should read this chapter first.
- **ZigBee Fundamentals:** Describes the characteristics of the ZigBee solution, and discusses a variety of topics, including a summary of the decisions to be made when designing a ZigBee solution.
- **Fundamental Design Choices:** Discusses the major decisions that must be made about how to architect a solution.
- **Introducing the Ember HAL:** The first half of the chapter describes some of the basic aspects of the Ember HAL, and is recommended for anyone using EmberZNet PRO. If you need to modify the HAL or port it to a new hardware platform, you should read the entire chapter.
- **Security Fundamentals:** 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.
- **Bootloading:** Introduces bootloading for Ember ZigBee networking devices. It looks at the concepts of standalone and application bootloading and discusses their relative strengths and weaknesses. In addition, it looks at design and implementation details for each method.
- **The Token System:** Describes tokens and shows how to use them in code. The chapter also discusses bindings, the application-defined associations between two devices on a network.
- **Application Development Tools:** Provides an overview of the toolchain used to develop, build, and deploy your applications, and discusses some additional tools and utilities.

An appendix defines abbreviations and acronyms used throughout the manual and in other related Ember documents.

1.3 Audience

This manual is intended as a starting place for anyone needing an introduction to developing ZigBee applications, or who are new to the Ember development environment.

1.4 Related Documentation

Individual chapters in this manual may point to more detailed documentation on their specific topic.

For a general introduction to testing and debugging Ember applications, see Ember document 120-3030-000, *Testing and Debugging Ember Applications*. For an overview of manufacturing testing, see Ember document 120-5074-000, *Manufacturing Test Overview*.

If you are developing a ZigBee-compliant application, additional information about Ember's AppBuilder tool and Application Framework can be found in InSight Desktop online help as well as Ember document 120-3028-000, *Application Framework Developer Guide*. AppBuilder is a tool for generating ZigBee-compliant applications, and provides a graphical interface for turning on or off embedded clusters and features in the code. The Application Framework is a superset of all source code needed to develop any ZigBee-compliant device.

If your application requires functionality not available through AppBuilder and the Application Framework, see Ember document 120-3031-000, *Advanced Application Programming with the Stack and HAL APIs*.

1.5 Getting Help

Development kit customers are eligible for training and technical support. You can use the Ember web site www.ember.com to obtain information about all Ember products and services, and to sign up for product support. You can also contact Ember Customer Support at http://www.ember.com/support_index.html.

2 Fundamentals of Wireless Networking

This chapter introduces some fundamental concepts of wireless networking. These concepts are referred to in other chapters of this guide. If you are new to wireless networking, you should read this chapter first.

2.1 Overview

As embedded system design has evolved, the need for networking support has become a basic design requirement. Like more general-purpose computers, embedded systems have moved toward wireless networking. Most wireless networks have pushed toward ever-higher data rates and greater point-to-point ranges. But not all design applications require high-end wireless networking capabilities. Low-data-rate applications have the potential to outnumber the classic high-data-rate wireless networks world wide. Simple applications such as lighting control, smart utility meters, HVAC control, fire/smoke/CO alarms, remote doorbells, humidity monitors, energy usage monitors, and countless others devices function very well with low-data-rate monitoring and control systems. The ability to install such devices without extensive wiring decreases installation and maintenance costs. Increased efficiencies and cost savings are the primary motives behind this applied technology.

A wireless sensor network (WSN) is a wireless network consisting of distributed devices using sensors at different locations to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion, or pollutants.

In addition to one or more sensors, each node in a sensor network is typically equipped with a radio transceiver or other wireless communications device, a small microcontroller, and an energy source, usually a battery. The size of a single sensor node can vary from shoebox-sized nodes down to devices the size of coins. The cost of sensor nodes is similarly variable, ranging from hundreds of dollars to a few dollars, depending on the size of the sensor network and the complexity required of individual sensor nodes. Size and cost constraints on sensor nodes result in corresponding constraints on resources such as energy, memory, computational speed, and bandwidth.

Wireless personal area networks have emerged as a result of the IEEE 802.15.4 standard for low data rate digital radio connections between embedded devices. The ZigBee Alliance was formed to standardize industry efforts to supply technology for networking solutions that are based on 802.15.4, have low data rates, consume very little power, and are therefore characterized by long battery life. The ZigBee Standard makes possible complete and cost-effective networked homes and similar buildings where all devices are able to communicate for monitoring and control.

2.2 Embedded Networking

While the term *wireless network* may technically be used to refer to any type of network that functions without the need for interconnecting wires, the term most commonly refers to a telecommunications network, such as a computer network. Wireless telecommunications networks are generally implemented with radios for the carrier or physical layer of the network.

One type of wireless network is a wireless local area network, or LAN. It uses radio instead of wires to transmit data back and forth between computers on the same network. The wireless LAN has become commonplace at hotels, coffee shops, and other public places. The wireless personal area network (WPAN) takes this technology into a new area where the distances required between network devices is relatively small and data throughput is low.

In the control world, embedded systems have become commonplace for operating equipment using local special-purpose computer hardware. Wired networks of such devices are now common in manufacturing environments and other application areas. Like all computer networks, the interconnecting cable systems and supporting hardware are messy, costly, and sometimes difficult to install. To overcome these problems, wireless networking of embedded systems (that is, embedded networking) has become commonplace. However, the costly embedded networking solutions have only been justifiable in high-end applications where the costs are a secondary consideration. Low cost applications with low data rate communications requirements did not have a good standardized solution until the IEEE 802.15.4 standard for wireless personal area physical layer and medium access control layer (MAC) was released in 2003.[1]

The ZigBee Alliance was formed to establish networking and application-level standards on top of the IEEE 802.15.4 standards, to allow flexibility, reliability, and interoperability. More recently, working groups (WG) have been formed within the Internet Engineering Task Force (IETF) to establish open standard approaches for routing (roll WG) and interfacing low-powered wireless devices to an IPV6 networks (6lowPan WG). Once these IETF working groups establish the standards, the role of the ZigBee Alliance shifts from setting standards to standards selection, compliance testing, and marketing.

Although wireless networks eliminate messy cables and enhance installation mobility, the downside is the potential for interference that might block the radio signals from passing between devices. This interference may be from other wireless networks or from physical obstructions that interfere with the radio communications. Interference from other wireless networks can often be avoided by using different channels. ZigBee, for example, has a channel-scanning mechanism on start-up of a network to avoid crowded channels. Standards-based systems, such as ZigBee and WiFi, use mechanisms at the medium access control layer to allow channel sharing. In addition, ZigBee provides an interoperable standard for multi-hop wireless networking, allowing signals to reach their destination by traveling through multiple relay points. ZigBee networks can be comprised of many such relay points or “routers”, each one within range of one or more other routers, creating an interconnected “mesh” of devices that can provide redundant paths for data within the network that are automatically rediscovered and used to avoid interference in a local area. This concept is collectively referred to as “mesh networking”.

[1] The current version as of this writing is the IEEE 802.15.4-2006 standard.

Another potential problem is that wireless networks may be slower than those that are directly connected through a cable. Yet not all applications require high data rates or large data bandwidth. Most embedded networks function very well at reduced throughputs. Application designers need to ensure their system data rates are within the range achievable with the system being used.

Wireless network security is also a problem, since the data can easily be overheard by eavesdropping devices. ZigBee has a set of security services designed around AES 128 encryption, so that application designers have a choice of security levels based on the needs of their applications. Careful design around these standards helps maintain high levels of network security.

Other networking standards exist such as *Bluetooth*. Each standard has its own unique strengths and essential areas of application. In the case of Bluetooth and ZigBee, the bandwidth of Bluetooth is 1 Mbps, while ZigBee's is one-fourth of this value. The strength of Bluetooth lies in its ability to allow interoperability and replacement of cables. ZigBee's strength is low cost, long battery life, and mesh networks for large network operation. Bluetooth is meant for point-to-point applications such as handsets and headsets, whereas ZigBee is focused on the sensors and remote controls market, large distributed networks, and highly reliable mesh networking.

Chapter 3 provides an in-depth discussion of the ZigBee Alliance, its efforts to standardize IEEE 802.15.4-based applications, and the characteristics of a ZigBee solution.

2.3 Radio Fundamentals

Radio is the wireless transmission of signals by modulation of electromagnetic waves with frequencies below those of visible light. Electromagnetic waves are, in the case of radio, a form of non-ionizing radiation, which travels by means of oscillating electromagnetic fields that pass through electrical conductors, the air, and the vacuum of space. Electromagnetic radiation does not require a medium of transport like a sound wave. Information can be imposed on electromagnetic waves by systematically changing (modulating) some property of the radiated waves, such as their amplitude or their frequency. When radio waves pass an electrical conductor, the oscillating fields induce an alternating current in the conductor. This can be detected and transformed into sound or other signals that reproduce the imposed information.

The word '*radio*' is used to describe this phenomenon and radio transmission signals are classed as *radio frequency emissions*. The range or spectrum of radio waves used for communication has been divided into arbitrary units for identification. The FCC and NTIA arbitrarily define the radio spectrum in the United States as that part of the natural spectrum of electromagnetic radiation lying between the frequency limits of 9 kilohertz and 300 gigahertz, divided into various sub-spectrums for convenience.

The following names are commonly used to identify the various sub-spectrums:

3 kHz to 30 kHz	Very Low Frequencies (VLF)
30 kHz to 300 kHz	Low Frequencies (LF)
300 kHz to 3,000 kHz	Medium Frequencies (MF)
3,000 kHz to 30,000 kHz	High Frequencies (HF)
30,000 kHz to 300,000 kHz	Very High Frequencies (VHF)
300,000 kHz to 3,000,000 kHz	Ultra High Frequencies (UHF)
3,000,000 kHz to 30,000,000 kHz	Super High Frequencies (SHF)
30,000,000 kHz to 300,000,000 kHz	Extremely High Frequencies (EHF)

Each of the sub-spectrums listed above are further subdivided into many other sub-portions or 'bands.' For example, the American AM Broadcast Band extends from 535 kHz to 1705 kHz, which is within the portion of the spectrum classified as *Medium Frequencies*.

2.3.1 Frequency Bands

The radio spectrum is regulated by government agencies and by international treaties. Most transmitting stations, including commercial broadcasters, military, scientific, industrial, and amateur radio stations, require a license to operate. Each license typically defines the limits of the type of operation, power levels, modulation types, and whether the assigned frequency bands are reserved for exclusive or shared use. Three frequency bands can be used for transmitting radio signals without requiring licensing from the United States Government:

900 MHz: The 900 MHz band was used extensively in different countries for different products including pages and cellular devices.

This band was considered to have good range characteristics. However it can be less popular for products because it is not a worldwide unlicensed band, and products therefore need to be modified depending on where they are being used.

2400 MHz: The 2400 MHz band is a very commonly-used frequency band. This band was one of the first worldwide unlicensed bands and therefore became popular for wireless consumer products.

Typical wireless technologies that use this band are 802.11b (1-11 Mbps), 802.11g (1-50 Mbps) and 802.15.4, as well as numerous proprietary radio types.

5200-5800 MHz: The 5200 MHz band has three sub-bands, the lowest being for indoor home use only, while the 5800 MHz frequencies can be used for long distance wireless links at very fast speeds (30 – 100 Mbps).

A common strategy is to use 2400 MHz in residential and home environments. The ZigBee Standard endorses the use of this band.

2.3.2 Signal Modulation

Modulation is the process of changing the behavior of a signal so that it transfers information. Modulation can also be thought of as a way to encode information to be

transmitted to a receiver that decodes, or demodulates, the information into a useful form.

The basic radio frequency (RF) signal has a fundamental frequency that can be visualized as an alternating current whose frequency is referred to as the *carrier wave frequency*. The earliest method used for encoding information onto the carrier wave involved switching the carrier wave on and off in a specific time duration pattern. This was known as continuous wave (CW) mode. The carrier frequency can also be varied in its amplitude (that is, signal strength) or its frequency. These two modulation methods are called amplitude modulation (AM) and frequency modulation (FM), respectively. It is possible to impose a signal onto the carrier wave using these three basic modulation techniques and creative variations of these techniques.

The EM2xx and EM35x use a form of *offset quadrature phase-shift keying* (OQPSK) to modulate the carrier wave. Phase-shift keying (PSK) is a digital modulation scheme that conveys data by changing, or modulating, the phase of a reference signal such as the carrier wave. PSK is a derivative of FM techniques.

All digital modulation schemes use a finite number of distinct signals to represent digital data. In the case of PSK, a finite number of phases are used. Each of these phases is assigned a unique pattern of binary bits. Usually, each phase encodes an equal number of bits. Each pattern of bits forms the symbol that is represented by the particular phase. The demodulator, which is designed specifically for the symbol set used by the modulator, determines the phase of the received signal and maps it back to the symbol it represents, thus recovering the original data. To do so the receiver must compare the phase of the received signal to a reference signal. Such a system is termed *coherent*.

2.3.3 Antennas

An antenna (or aerial) is an arrangement of electrical conductors designed to emit or capture electromagnetic waves. The ability of an antenna to emit a signal that can be detected by another antenna is referred to as *radio propagation*. Antennas are made to a certain size based on the operating frequencies. An antenna from a 2400 MHz radio cannot be used effectively on a 5800 MHz radio, or vice versa. However, an antenna from one type of 2400 MHz technology, such as WiFi or Bluetooth, can be used in another 2400 MHz technology, such as ZigBee.

Two fundamental types of antennas are described with reference to a specific three dimensional space:

- Omni-directional (radiates equally in all directions)
- Uni-directional (also known as *directional*) (radiates more in one direction than in the other). All antennas radiate some energy in all directions in free space, but careful construction results in substantial transmission of energy in certain directions and negligible energy radiated in other directions.

Because of the nature of mesh networking, in general an omni-directional antenna is desired to provide as many communication paths as possible.

2.3.4 How Far Signals Travel

The distance a radio signal will travel and the amount of information that can be transmitted is based on:

- The amount of power the antenna is transmitting into the air.
- The distance between the transmitting and receiving stations.

- How much radio signal strength the receiving radio needs.
- What types of physical/electrical obstructions are in the way.

2.3.4.1 Radio Transmit Power

Radio transmit power is measured in decibels (dB), and sometimes is discussed in terms of *watts*. Converting wattage to dB allows radio link calculations using simple addition and subtraction.[2]

For example, a typical power amplified wireless radio card transmits at 100 milliwatts (or mW), which translates to a power output of 20 dBm.

If 1 mW, or 0 dBm, is the baseline for power in decibels, then +3 dBm is some power level above 1 mW (2 mW to be specific). This is the standard output power of the EM250/EM260/EM351/EM357 devices. In Boost Mode, that can be increased to +5 dBm, or 3.16 mW, on the EM2xx platforms and +8 dBm, or 6.3 mW, on the EM35x platforms. Using a power amplifier module can increase the transmit power to 20 dBm, but this requires more power to operate.

2.3.4.2 Signal Degradation

The radio also needs to be able to *hear* a radio signal at a certain level. The minimum signal strength required for a receiver to understand the data is called the *receive sensitivity*.

As the radio signal travels through the air, it weakens. When a radio signal leaves the transmitting antenna the dB is a high number (for example: 20 dB). As it travels through the air, it loses strength and drops to a negative number. At some point, a minimum value for dB is reached, below which the radio will no longer successfully receive the transmission. This value represents the "receive sensitivity" or "RX sensitivity". This value will vary with the type of radio used but is typically between -90 dB and -100 dB. Refer to the datasheet for your radio chip for specific receive sensitivity figures.

If you can achieve a signal level of -75 dB and your radio has an Rx sensitivity of -95 dB, you have 20 dB of extra signal to accommodate interference and other issues. This is called *margin*.

2.3.4.3 How Far Can the Radio Signal Go

If you know the power out, and the receiver sensitivity, you can determine whether you can broadcast over a given distance. In the following example, you want to know if you can receive a signal over five miles. To do so, you need to know the free space loss between the radio transmitter and the receiver.

[2] dBm= 10*log10(P/ 0.001)

For example, free space loss of a 2.4 GHz signal at 5 miles is 118.36 dB. So, you can estimate signal strength over the range of the network as:

What	Add or subtract it	The value
Transmitter power	+	15 dBm
Transmitter antenna gain	+	14 dBi
Receiver antenna gain	+	14 dBi
Transmitter's coaxial cable loss	-	2 dB
Receiver's coaxial cable loss	-	2 dB
Free Space Loss @ 5 miles	-	118.36 dB
	Total:	-79.36 dB

In other words, a 15 dBm radio hooked into a 14 dBi antenna, transmitting 5 miles through free space to another radio hooked up to a 14 dBi antenna, yields approximately -79 dB of signal. However, physical obstructions such as buildings or trees would have a substantial impact on these calculations. Typical ZigBee networks use smaller, lower-cost antennas without the gain increase and only use power amplifiers if extended range is required. Occasionally, external low noise amplifiers (LNAs) may also be employed to boost RX sensitivity of incoming signals just before they reach the radio.

This calculation is provided as an example. The Ember development kits for the EM250/EM260/EM351/EM357 come with a functional test application ("nodetest") that can be used to perform empirical range testing for an embedded wireless network in virtually any environment (see Ember documents 120-5031-000, 120-5041-000, and 120-5064-000, *Bringing up Custom Devices for the EM250*, *EM260*, and *EM35x*, respectively, for information on using nodetest). Ember recommends that basic range testing be conducted in the expected environment to evaluate whether extended range is required.

2.4 Networking: Basic Concepts

A network is a system of computers and other devices (such as printers and modems) that are connected in such a way that they can exchange data. This data may be informational or command-oriented, or a combination of the two.

A networking system consists of hardware and software. Hardware on a network includes physical devices such as a computer workstations, peripherals, and computers acting as file servers, print servers, and routers. These devices are all referred to as *nodes* on the network.

If the nodes are not all connected to a single physical cable, special hardware and software devices must connect the different cables in order to forward messages to their destination addresses. A *bridge or repeater* is a device that connects networking cables without examining the addresses of messages or making decisions as to the best route for a message to take. In contrast, a *router* contains addressing and routing information that lets it determine, from a message's address, the most efficient route for the message. A message can be passed from router to router several times before being delivered to its target destination.

In order for nodes to exchange data, they must use a common set of rules defining the format of the data and the manner in which it is to be transmitted. A *protocol* is a formalized set of procedural rules for the exchange of data. The protocol also provides rules for the interactions among the network's interconnected nodes. A network software developer implements these rules in software applications that carry out the functions required by the protocol.

Whereas a router can connect networks only if they use the same protocol and address format, a gateway converts addresses and protocols to connect dissimilar networks. Such a set of interconnected networks can be referred to as an internet, intranet, wide area network (WAN), or other specialized network topology. The term Internet is often used to refer to the largest worldwide system of networks, also called the World Wide Web. The basic protocol used to implement the World Wide Web is called the Internet protocol, or IP.

A *networking protocol* commonly uses the services of another, more fundamental protocol to achieve its ends. For example, the Transmission Control Protocol (TCP) uses the Internet Protocol (IP) to encapsulate the data and deliver it over an IP network. The protocol that uses the services of an underlying protocol is said to be a client of the lower protocol; for example, TCP is a client of IP. A set of protocols related in this fashion is called a protocol stack.

2.5 Wireless Networking

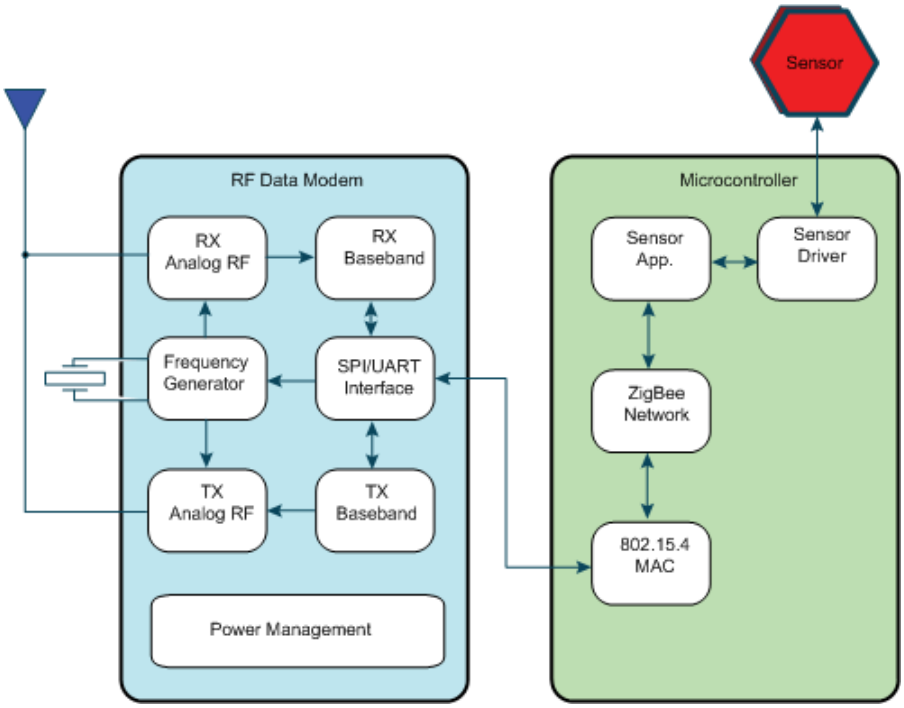
Wireless networking mimics the wired network, but replaces the wire with a radio signal as the data interconnection medium. Protocols are essentially the same as used in wired networks, although some additional functionality has been added, so that the two types of networks remain interoperable. However, wireless networks have emerged that do not have a wired counterpart requiring interoperability. These specialized networks have their own hardware and software[3] foundations to enable reliable networking within the scope of their unique environments.

2.6 Ember ZigBee Devices

Ember has developed networking hardware (the EM2xx and EM35x integrated circuit families) and software (the EmberZNet PRO stack and development tools) to facilitate implementation of a wireless personal area network of devices for sensing and control applications. The diagram in Figure 2-1 represents a typical wireless device using ZigBee technologies. The RF data modem is the hardware responsible for sending and receiving data on the network. The microcontroller represents the computer control element that originates messages and responds to any information received. The sensor block can be any kind of sensor or control device. Such a system can exist as a node on a ZigBee network without any additional equipment. Any two such nodes, with compatible software, can form a network. Large networks can contain thousands of such nodes.

[3]Most networking protocols are based, to some degree, on the Open Systems Interconnection (OSI) Model.

Figure 2-1. Typical ZigBee Device Block Diagram



The EM250/EM351/EM357 provide both the RF and microcontroller portions of Figure 2-1. The EM260 (and the EM35x chips when used as "network coprocessors") provide only the RF and networking part of the system, acting as a coprocessor to any microcontroller, DSP or similar device required for the application.

EmberZNet PRO networks support the device types listed in Table 2-1.

Table 2-1. EmberZNet PRO Node Types

Node type	Description
`EMBER_COORDINATOR` (ZigBee coordinator)	The coordinator initiates network formation, and functions as a routing device that relays messages and can act as a parent to other nodes. This device is normally always powered on.
`EMBER_ROUTER` (ZigBee router)	A full-function routing device that relays messages and can act as a parent to other nodes. These devices must be always powered on.
`EMBER_END_DEVICE` (ZigBee end device with `RXOffWhenIdle` flag cleared)	An end device whose radio is always on, but communicates only through its parent and will not relay messages.
`EMBER_SLEEPY_END_DEVICE` (ZigBee end device with `RXOffWhenIdle` flag set)	An end device whose radio can be turned off to save power.
`EMBER_MOBILE_END_DEVICE` (ZigBee end device with `RXOffWhenIdle` flag set)	An end device that can move through the network. The EmberZNet PRO stack always assumes a Mobile End Device is a sleepy node. Not a ZigBee standard device type.

Coordinator and router devices form the basis of the network and route data for other devices in the network.

End devices send and receive messages only from their *parent*. A parent is a router/coordinator in direct communication range of the "child" end device. The parent acts as a proxy for inbound and outbound traffic pertaining to that end device "child". This allows the end devices to sleep while their parent holds messages for them until they wake up. End devices do not relay messages for other devices.

2.6.1 Network Formation and Operation

The coordinator initiates network formation. In a mesh network, after the coordinator forms the network it can function as a router. The EmberZNet PRO libraries enable any device to act as a coordinator and form a network. After forming a network, the coordinator can accept requests from other devices that wish to join the network. Depending on the stack and application profile used, the coordinator might also perform additional duties after network formation. An application profile describes the messages and network setting for a particular application, such as smart energy. See Chapter 3 for more information.

A device finds a network by scanning channels. When a device finds a network with the correct profile that is open to joining, it can request to join that network. A device can send a join request to the network's coordinator or one of its router nodes. If the application is using a *trust center*, the trust center can further specify security conditions under which join requests are accepted or denied. See Chapter 6 for more information about security.

All nodes that communicate on a network transmit and receive on the same channel, or frequency. ZigBee uses a *personal area network identifier* (PAN ID) to identify a network. The PAN ID provides a way for two networks to exist on the same channel while still maintaining separate traffic flow. Note that when two networks exist in the same channel they have to share time on the air.

The network layer discovers and maintains available routes so that the user application does not need to know anything about the underlying routes used to deliver a message to a destination node. Route discovery varies among networks and routing mechanisms. The two general approaches are active and dynamic discovery:

- **Active route discovery** tries to keep certain routes up to date at all times. This consumes additional network overhead but means that routes are available whenever a node wishes to send data.
- **Dynamic, or on-demand, route discovery** incurs less overhead network traffic but can cause a delay when a route changes because of shifting radio conditions or network rearrangements.

In an EmberZNet PRO stack, after a route between a source node and target node is discovered, the source node sends the message to the first node in the route, as specified in the source node's routing table. Each intermediate node uses its own routing table to forward the message to the next node along the route, until the message reaches the target node. The information about the route is next-hop, where each node knows what the next hop should be for delivery to a particular destination. If a route fails, the source node must find a new route.

3 ZigBee Fundamentals

This chapter describes the characteristics of a ZigBee solution, and discusses:

- Network topologies
- Network node types
- Routing concepts
- The ZigBee stack
- The ZigBee Cluster Library (ZCL)
- ZigBee compliance
- ZigBee IP

It concludes with a summary of the decisions to be made when designing a ZigBee solution.

3.1 Introduction

ZigBee refers both to:

- An open standard for reliable, cost-effective, low power, wireless device-to-device communication
- An alliance of over 400 companies who together are defining and using the standard to communicate in a variety of applications such as smart energy and commercial building automation.

The ZigBee Alliance is operated by a set of promoter companies that make up the Board of Directors. These currently include Ember, ST Microelectronics, Philips, Schneider Electric, Landis and Gyr, Itron, Tendril, Emerson, Kroger, Reliant Energy, Texas Instruments, and Freescale. The Alliance activities are accomplished through workgroups dedicated to specific areas of the technology. These include a network group, a security group, an application profile group, and several others. Standards are available to members and then non-members for download. To use the ZigBee technology within a product, companies are required to become members of the Alliance.

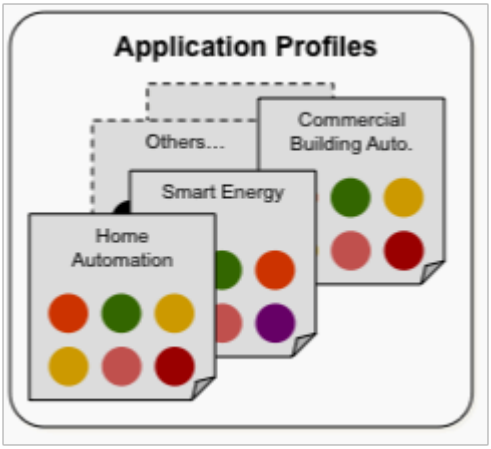
Ember is a founding member of the ZigBee Alliance. The Ember networking stack has been a Golden Platform for testing and certification for all new revisions of the standard to date. Ember is active in a variety of areas within the Alliance and with helping customers adopt ZigBee technology.

The ZigBee Alliance specifies three major items of interest.

1. **ZigBee networking** – The basic levels of network interaction, including acceptable RF behavior, methods of creating and joining the network, discovering routes, and using routes to transmit traffic over the network.

2. **ZigBee application profiles** – Describe a set of messages and network settings for a particular application. All devices adhering to these settings may interoperate. Application profiles may be public or private. Public profiles are published, approved by and distributed by the ZigBee Alliance, such as those shown in Figure 3-1. Private profiles are used by a single company or private groups of companies working together.

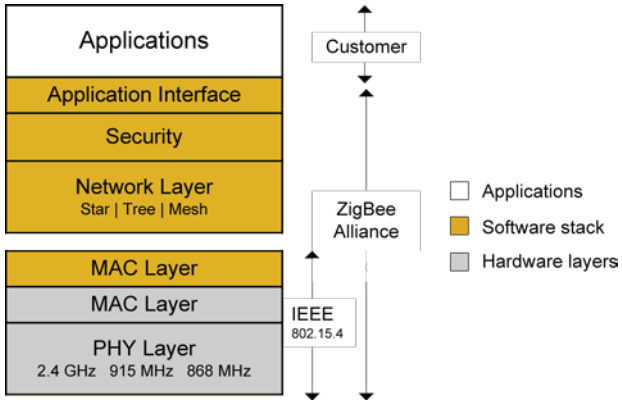
Figure 3-1. ZigBee Application Profiles



3. **Certification** – Applications using one or more public ZigBee application profiles are certified by the Alliance to indicate to end users that the device(s) are compliant. Applications using private profiles receive certification that they are using a properly operating ZigBee stack and will not adversely affect other ZigBee networks during operation.

Because ZigBee is committed to open and interoperable devices, standards have been either adopted where they existed, or developed, from the physical layer through the application layer, as shown in Figure 3-2. At the physical and MAC (medium access control) layer, ZigBee adopted the IEEE 802.15.4 standard. The networking, security and application layers have all been developed by the ZigBee Alliance. An ecosystem of supporting systems such as gateways and commissioning tools has also been developed to simplify the development and deployment of ZigBee networks. Extensions and additions to the standards continue to be developed, and Ember is committed to supporting these as they are available.

Figure 3-2. ZigBee Architecture



Ember provides a standard networking API based on the ZigBee specification across its different platforms (EM2xx and EM35x series SoCs and NCPs). Ember also provides tools which allow rapid ZigBee application development, including an Application Framework and a UI-based tool called AppBuilder.

The following three sections describe the general characteristics of a ZigBee network, discuss the use of IEEE 802.15.4 standard, and summarize the hardware and software elements of a ZigBee network.

3.1.1.1 General Characteristics

ZigBee is intended as a cost-effective and low power solution. It is targeted to a number of markets including home automation, building automation, sensor networks, smart energy, and personal health care monitoring. The general characteristics for a ZigBee network are as follows:

- Low power – Devices can typically operate for several years on AA type batteries using suitable duty cycles. With extremely careful design and special battery technologies, some ZigBee devices such as gas meters can achieve 20 years of battery life.
- Low data rate – The 2.4 GHz band supports a radio data rate of 250 kbps. Actual sustainable traffic through the network is lower than this theoretical radio capacity. As such, ZigBee is better used for sampling and monitoring applications or basic control applications.
- Small and large networks – ZigBee networks vary from several devices to thousands of devices communicating seamlessly. The networking layer is designed with several different data transfer mechanisms (types of routing) to optimize the network operation based on the expected use.
- Range – Typical devices provide sufficient range to cover a normal home. Readily available designs with power amplifiers extend the range substantially. A distributed spread spectrum is used at the physical layer to be more immune to interference.
- Simple network installation, start up and operation – The ZigBee standard supports several network topologies. The simple protocols for forming and joining networks allow systems to self configure and fix routing problems as they occur.

3.1.1.2 IEEE 802.15.4

ZigBee networks are based on the IEEE 802.15.4 MAC and physical layer. The 802.15.4 standard operates at 250 kbps in the 2.4 GHz band and 40 kbps/20 kbps in the 900/868 MHz bands. A number of chip companies provide solutions in the 2.4 GHz band with a smaller number supporting the 900/868 MHz band. ZigBee PRO uses 802.15.4-2003. ZigBee IP uses the 802.15.4 – 2006 version of the standard.

The 802.15.4 MAC layer is used for basic message handling and congestion control. This MAC layer includes mechanisms for forming and joining a network, a CSMA mechanism for devices to listen for a clear channel, as well as a link layer to handle retries and acknowledgement of messages for reliable communications between adjacent devices. The ZigBee network layer builds on these underlying mechanisms to provide reliable end to end communications in the network. The 802.15.4 standard is available from www.ieee.org.

The 802.15.4 standard provides some options within the MAC layer on beacon networks, guaranteed time slots that are not used by ZigBee in any current stack profiles. As such, these items are not normally included in the ZigBee software stack to save code space. ZigBee also has made specific changes to the 802.15.4 MAC that are documented in Annex D of the ZigBee specification.

3.1.1.3 Hardware and Software Elements

A ZigBee solution requires implementation of a ZigBee radio and associated microprocessor (together in a single chip or separately), and implementation of an application on top of a ZigBee stack.

Typically a developer can purchase a ZigBee radio and software as a bundled package, although some third party software stacks have been developed. In general, the hardware and software provider includes reference designs for the hardware and sample applications for the software. Based on these, hardware developers can customize the hardware to their specific needs. Alternatively, a number of module providers can deliver compact and low cost custom modules.

Because of the embedded nature of typical ZigBee applications, software application development is typically interrelated with the hardware design to provide an optimal solution. Ember offers the Application Framework, which gives customers a way to rapidly develop their applications based on ZigBee application profiles and the ZigBee cluster library (ZCL). Alternatively, a number of third party software development firms specialize in developing ZigBee applications and can assist in new product development.

The Ember AppBuilder tool works with the Application Framework, and includes support for a number of ZigBee profiles (such as Home Automation (HA), Smart Energy (SE), and ZigBee Light Link (ZLL)). Ember also offers a number of utilities. These include:

- Over-the-air bootloaders to allow upgrades to system software after deployment
- Gateway interfaces to interface the ZigBee network to other systems
- Programming tools for the microprocessor
- Network sniffer and debug tools to allow viewing and analysis of network operations.
- Processor debug tools through partnerships with leading firms.

3.2 ZigBee Network Topologies

ZigBee supports two basic topology types:

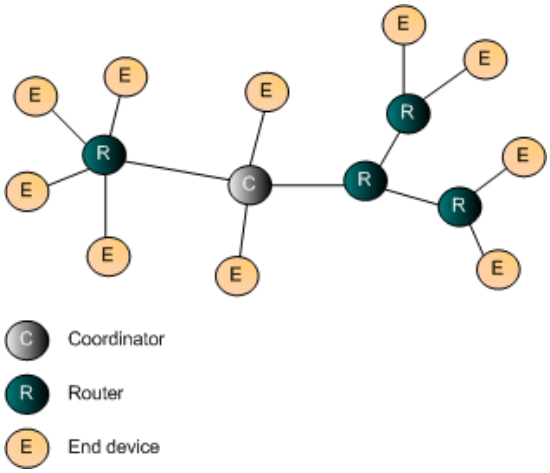
- Tree Network
- Mesh Network

The actual topology used should be based on the network design, the individual devices that compose the network, and the data expected to flow within the system.

3.2.1 Tree network

The ZigBee tree topology, as shown in Figure 3-3, is used for address assignment and provides a routing method. Routers branch out in a tree-like fashion from the coordinator, and end devices are potential sleepy devices. The ZigBee specification supports a mesh overlaid on the tree, also known as table routing. EmberZNet does not support a tree topology due to fault recovery issues, and instead always uses the mesh topology described in the next section.

Figure 3-3. Tree Network



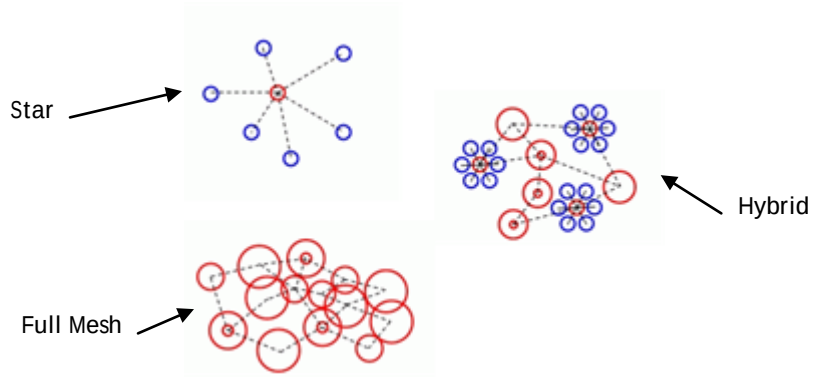
3.2.2 Mesh network

Embedded mesh networks make radio systems more reliable by allowing radios to relay messages for other radios. For example, if a node cannot send a message directly to another node, the embedded mesh network relays the message through one or more intermediary nodes.

EmberZNet PRO supports three types of mesh network topologies, as shown in Figure 3-4:

- Star Network
- Full Mesh Network
- Hybrid Mesh Network

Figure 3-4. Star, Full Mesh, and Hybrid Mesh Networks



Note: Blue devices in Figure 3-3 are end devices and can be sleepy or mobile. See Section 3.3, Network Node Types, for more information about end devices.

3.2.2.1 Star network

In a star network, one hub is the central point of all communications. The hub can become bottlenecked with network/processing bandwidth. This topology is not very mesh-like, and transmission is limited by the hub's communication radius. Outlying nodes can be battery powered. In the EmberZNet stack, this topology is formed by a group of end devices with a coordinator node as their parent. The coordinator node serves as network hub.

3.2.2.2 Full mesh network

In a full mesh network, all nodes are router nodes, including the coordinator after it forms the network. Because all nodes can relay information for all other nodes, this topology is least vulnerable to link failure; it is highly unlikely that one device might act as a single point of failure for the entire network.

3.2.2.3 Hybrid mesh network

A hybrid mesh network topology combines star and mesh strategies. Several star networks exist, but their hubs can communicate as a mesh network. A hybrid network allows for longer distance communication than a star topology and more capability for hierarchical design than a mesh topology. This topology is formed by the EmberZNet stack, using router devices as hubs for the star subnetworks, where each hub can have end devices attached to it.

The choice of topology must take into consideration which nodes are line-powered or battery-powered, expected battery lifetime, amount of network traffic required, latency requirements, the cost of the solution, as well as other factors. See Chapter 4, Fundamental Design Choices, for more information.

3.3 Network Node Types

The ZigBee specification supports networks with one coordinator, multiple routers, and multiple end devices within a single network. These node types are described in the following sections.

3.3.1 Coordinator

A ZigBee coordinator (ZC) is responsible for forming the network. A coordinator is a router with some additional functionality. ZigBee coordinator functions include selecting an appropriate channel after scanning available channels, and selecting an extended PAN ID (See section 3.5.4 for more information about extended PAN IDs). After forming the network, the coordinator acts as a router. If the network profile does not use mesh routing, the coordinator plays a crucial role as the root of the tree in the tree routing algorithm.

For some applications, the coordinator may have added responsibilities, such as being the trust center or network manager. The trust center manages the security settings and authorizations for the network. The network manager monitors and corrects network issues such as PAN ID conflicts or channel changing due to interference. These choices are up to the application developer, and in some cases are made by the appropriate ZigBee stack profile. For example under the Home Automation application profile the coordinator is always the trust center for the network.

Note: Only a network coordinator can be designated as a security trust center when starting a network.

3.3.2 Routers

ZigBee router devices (ZR) provide routing services to network devices. Routers can also serve as end devices. Unlike end devices, routers are not designed to sleep and should generally remain on as long as a network is established.

3.3.3 End devices

End devices (ZED) are leaf nodes. They communicate only through their parent nodes and, unlike router devices, cannot relay messages intended for other nodes.

Depending on the network stack, end devices can be of several types:

- Sleepy end devices (EmberNodeType EMBER_SLEEPY_END_DEVICE) power down their radio when idle, and thus conserve resources. However, they must poll their parent node to receive incoming messages and acknowledgements; no data is sent to the sleepy end device until the end device requests it. Sleepy end devices are also sometimes known as rx-off-when-idle devices. This is a standard ZigBee device type.
- Non-sleepy end devices (EmberNodeType EMBER_END_DEVICE) do not route messages for other devices but they remain powered during operation. These devices are known as Rx-on-when-idle devices. This is a standard ZigBee device type.

- Mobile end devices (EmberNodeType EMBER_MOBILE_END_DEVICE) is a sleepy end device with enhanced capabilities that enable it to change its physical location and quickly switch to a new parent router. This device type is an Ember modification to the basic ZigBee sleepy end device to provide added capabilities and management of mobile devices.

The EmberZNet PRO stack supports sleepy and non-sleepy end devices. The choice of node type must also be considered carefully. For example, in a very dense network, it is not always advantageous for all line powered nodes to be routers due to possible interference issues which may occur when a child node tries to find a parent node to communicate with. It's important to try to create a balanced network where all nodes have redundant paths, but where there aren't too many routers in close proximity to create interference.

3.4 ZigBee Routing Concepts

3.4.1 Overview

ZigBee has several routing mechanisms that can be used based on the network and expected traffic patterns.[1] The application designer should choose which mechanism to use as part of the system architecture and design. In actual practice one application may use several of these routing mechanisms, because some devices may be performing one-to-one communications while other devices may be communicating to a central monitoring device. The types of routing discussed below are:

- tree routing
- table routing
- broadcasts
- multicast routing,
- many-to-one/source routing.

The ZigBee and ZigBee PRO stacks have different routing mechanisms.

Tree Routing

In the ZigBee stack (as opposed to the ZigBee PRO stack), routing is initially done along the links of the network's tree topology, although this route might be indirect. For example, two nodes might be located next to each other but at the same depth of the network tree—that is, they are the same number of hops away from the coordinator node. If they join the network through different parent nodes, tree routing of messages to one another occurs by passing the message up the tree as many levels as necessary until they find a common ancestor node. This tree routing mechanism assumes that the network tree topology is always stable—tree paths never change, so discovery is not required. Routes are deterministic and can be calculated mathematically. Note that the EmberZNet stack does not support the tree topology due to inherent failover issues found in the ZigBee stack implementation of tree routing.

Table Routing

The ZigBee PRO stack never uses tree routing for message delivery because an alternate addressing schemes is used and the tree does not exist in the network. Routes are

[1] See ZigBee Specification, document 053474. Section 3.5.8 describes the frame format, and section 3.7.3.4 describes the behavior.

formed when one node sends a route request to discover the path to another node. After a route is discovered between the two nodes, the source node sends its message to the first node in the route, as specified in the source node's routing table. Each intermediate node uses its own routing table to forward the message to the next node (that is hop) along the route until the message reaches its destination. If a route fails, a route error is sent back to the originator of the message who can then rediscover the route.

Broadcast Routing

Broadcast routing is a mechanism to send a message to all devices in a network. Network-level broadcast options exist to send to routers only or also to send to sleeping end devices. A broadcast message is repeated by all router-capable devices in the network three times to ensure delivery to all devices. While a broadcast is a reliable means of sending a message, it should be used sparingly because of the impact on network performance. Repeated broadcasts can limit any other traffic that may be occurring in the network. Broadcasts are also not a reliable means of delivery to a sleeping device because the parent device is responsible for buffering the message for the sleeping child but may drop the message before the end device wakes to receive it.

Multicast Routing

Multicast routing provides a one-to-many routing option. A multicast is used when one device wants to send a message to a group of devices, such as a light switch sending an on command to a bank of 10 lights. Under this mechanism, all the devices are joined into a multicast group. Only those devices that are members of the group will receive messages, although other devices will route these multicast messages. A multicast is a filtered limited broadcast. It should be used only as necessary in applications, because over-use of broadcast mechanisms can degrade network performance. A multicast message is never acknowledged.

Many-to-One/Source Routing

Many-to-one routing is a simple mechanism to allow an entire network to have a path to a central control or monitoring device. Under normal table routing, the central device and the devices immediately surrounding it would need routing table space to store a next hop for each device in the network, as well as an entry to the central device itself. Given the memory limited devices often used in ZigBee networks, these large tables are undesirable.

Under many-to-one routing, the central device, known as a “concentrator,” sends a single route discovery that established a single route table entry in all routers to provide the next hop to the central device. This yields a result similar to that of table routing, but with a single many-to-one route request rather than many individual, one-to-one route requests from each router towards the concentrator.

All devices in the network then have a next hop path to the concentrator and only a single table entry is used. However, often the central device also needs to send messages back out into the network. This would result in a more significant increase in route table size, particularly for those nodes closest to the concentrator, since they are relay points in the concentrator’s many outbound routes to the rest of the network. Instead, incoming messages to the concentrator first use a route record message to store the sequence of hops used along the route. The concentrator then stores these next hop routes in reverse order as “source routes” in a locally held table known as a “source route table”. Outgoing messages include this source route in the network header of the message. The message is then routed using next hops from the network header instead of from the route table. This provides for large scaleable networks without increasing the memory requirements of all devices. It should be noted that the concentrator does require some additional memory if it is storing these source routes.

For detailed information on message delivery, refer to the ZigBee specification available from www.zigbee.org.

3.4.2 Using Link Quality to Aid in Routing

The information in this section is provided for those wishing to understand the details of the network layer's operation, which can prove useful during troubleshooting. Otherwise, link status messages are handled automatically by the stack and application writers need not be concerned with it.

Links in wireless networks often have asymmetrical link quality due to variations in the local noise floor, receiver sensitivity, and transmit power. The routing layer must use knowledge of the quality of links in both directions in order to establish working routes and to optimize the reliability and efficiency of those routes. It can also use the knowledge to establish reliable two-way routes with a single discovery.

ZigBee routers (ZR) keep track of inbound link quality in the neighbor table, typically by averaging LQI (Link Quality Indicator) measurements made by the physical layer. To handle link asymmetry, the ZigBee PRO stack profile specifies that routers obtain and store costs of outgoing links as measured by their neighbors. This is accomplished by exchanging link status information through periodic one hop broadcasts, referred to as "link status" messages. The link status algorithm is explained below, as implemented in EmberZNet PRO.

3.4.2.1 Description of Relevant Neighbor Table Fields

ZigBee routers store information about neighboring ZigBee devices in a neighbor table. For each router neighbor, the entry includes the following fields:

- average incoming LQI
- outgoing cost
- age

The incoming LQI field is an exponentially weighted moving average of the LQI for all incoming packets from the neighbor. The incoming cost for the neighbor is computed from this value using a lookup table.

The outgoing cost is the incoming cost reported by the neighbor in its neighbor exchange messages. An outgoing cost of 0 means the cost is unknown. An entry is called "two-way" if it has a nonzero outgoing cost, and "one-way" otherwise.

The age field measures the amount of time since the last neighbor exchange message was received. A new entry starts at age 0. The age is incremented every `EM_NEIGHBOR_AGING_PERIOD`, currently 16 seconds. Receiving a neighbor exchange packet resets the age to `EM_MIN_NEIGHBOR_AGE`, as long as the age is already at least `EM_MIN_NEIGHBOR_AGE` (currently defined to be 3). This makes it possible to recognize nodes that have been recently added to the table and avoid evicting them, which reduces thrashing in a dense network. If the age is greater than `EM_STALE_NEIGHBOR` (currently 6), the entry is considered stale and the outgoing cost is reset to 0.

3.4.2.2 Link Status Messages

Routers send link status messages every 16 seconds plus or minus 2 seconds of jitter. If the router has no two-way links it sends them eight times faster. The packet is sent as a one-hop broadcast with no retries. In the EmberZNet stack, they are sent as ZigBee network command frames.

The payload contains a list of short IDs of all non-stale neighbors, along with their incoming and outgoing costs. The incoming cost is always a value between 1 and 7. The outgoing cost is a value between 0 and 7, with the value 0 indicating an unknown outgoing cost. For frame format details, refer to the ZigBee specification. Link status messages are also automatically decoded by InSight Desktop for easy reference.

Upon receipt of a link status message, either a neighbor entry already exists for that neighbor, or one is added if there is space or if the neighbor selection policy decides to replace an old entry with it. If the entry does not get into the table, the packet is simply dropped. If it does get in, then the outgoing cost field is updated with the incoming cost to the receiving node as listed in the sender's neighbor exchange message. If the receiver is not listed in the message, the outgoing cost field is set to 0. The age field is set to EM_MIN_NEIGHBOR_AGE.

3.4.2.3 How Two-way Costs are Used by the Network Layer

As mentioned above, the routing algorithm makes use of the bidirectional cost information to avoid creating broken routes, and to optimize the efficiency and robustness of established routes. For the reader familiar with the ZigBee route discovery process, this subsection gives details of how the outgoing cost is used. The mechanism is surprisingly simple, but provides all the benefits mentioned above.

Upon receipt of a route request command frame, the neighbor table is searched for an entry corresponding to the transmitting device. If no such entry is found, or if the outgoing cost field of the entry has a value of 0, the frame is discarded and route request processing is terminated.

If an entry is found with non-zero outgoing cost, the maximum of the incoming and outgoing costs is used for the purposes of the path cost calculation, instead of only the incoming cost. This value is also used to increment the path cost field of the route request frame prior to retransmission.

3.4.2.4 Key Concept: Rapid Response

Rapid response allows a node that has been powered on or reset to rapidly acquire two-way links with its neighbors, minimizing the amount of time the application must wait for the stack to be ready to participate in routing. This feature is 100% ZigBee compatible.

If a link status message is received that contains no two-way links, and the receiver has added the sender to its neighbor table, then the receiver sends its own link status message immediately in order to get the sender started quickly. The message is still jittered by 2 seconds to avoid collisions with other rapid responders. To avoid a chain reaction, rapid responders must themselves have at least one two-way link.

3.4.2.5 Key Concept: Connectivity Management

By nature ZigBee devices are RAM-constrained, but often ZigBee networks are dense. This means that each router is within radio range of a large number of other routers. In such cases, the number of neighbors can exceed the maximum number of entries in a device's neighbor table. In such cases, the wrong choice of which neighbors to keep can lead to routing inefficiencies or worse — a disconnected network. EmberZNet employs 100% ZigBee-compatible algorithms to manage the selection of neighbors in dense networks to optimize network connectivity.

3.4.3 Route Discovery & Repair

Routing in ZigBee is automatically handled by the networking layer, and the application developer usually does not need to be concerned with its behavior. However, it is useful to have a feel for how the network behaves when a route needs to be discovered or repaired.

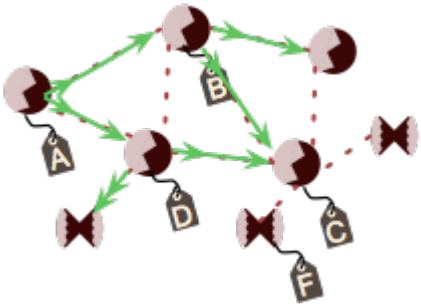
3.4.3.1 Route discovery

Route discovery is initiated when a unicast message is sent from one device to another and there is no pre-existing route.

We assume that there is no existing route so the networking software will begin the process of route discovery. For simplicity, assume that the routing tables of all devices are blank.

For example, assume that Devine A needs to send a message to device C, as shown in Figure 3-5. Device A will broadcast a message to the entire network asking the device C to reply. This broadcast message also serves to establish a temporary route back to A, as each intermediate device records the device from which it received the message. Routes are updated on intermediate nodes — note that these are temporary entries that have a shorter lifetime than regular entries and are not intended for re-use. Because A is a one-hop neighbor, B and D do not need to store routing information about it.

Figure 3-5. Example Network



C could use either B or D as its next hop back to A. ZigBee leaves this choice to the implementation; Ember uses a weighting algorithm to choose the most apparently reliable next hop.

When the message reaches device C, C sends a special unicast message (called a Route Response message) back to A using the temporary route constructed in step 1. This message is used by intermediate devices to establish a (permanent) route back to C.

Because C is a one-hop neighbor, B does not need to store routing information about it. D is not involved in this part of the discovery process because it was not selected by A in the above step. When the message reaches device A, the route discovery is complete and the new route can be used to send data messages from A to C.

ZigBee PRO networks will detect asymmetric RF links and avoid them during route discovery. This improves the reliability of the discovery process and the resulting routes.

Routes that have not been used within a certain timeout period (1 minute in EmberZNet 3.0 and later) are marked for re-use and new routes may then overwrite that memory

location. In some cases a new route may be needed and one or more intermediate devices will not have an available routing table entry; in this case the message will be reported as undeliverable to the sending node.

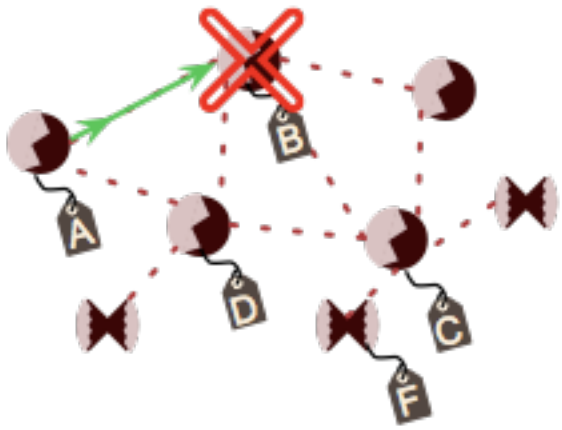
The application specifies if an end-to-end acknowledgement should be sent by the receiver (this is called an APS acknowledgement). If yes, the sender will be notified upon successful delivery of in the case of a timeout waiting for acknowledgement. In the case of a timeout, the route may need to be repaired.

Figure 3-6. Unicast message (green) with acknowledgement (blue)

3.4.3.2 Route repair

When a unicast message is sent with an acknowledgement requested, the sending device will be informed when the message is successfully delivered. If does not receive this acknowledgement, it can then take steps to repair the route. Route repair follows exactly the same steps as route discovery, above, but the damaged node (B, in Figure 3-7) does not participate, resulting in a different route choice.

Figure 3-7. Network with a Damaged Node



The routing table for A is updated to reflect that the next hop is D and the message is successfully delivered along the new path, as shown in Figure 3-8.

Figure 3-8. Alternative Route

In the case that no alternative path is available, the sender will be informed that the message could not be delivered. In EmberZNet PRO this is a 0×66 EMBER_DELIVERY_FAILED response.

EmberZNet PRO will attempt to deliver a message again before performing the route repair. Route repair is performed automatically when EMBER_APS_OPTION_RETRY and EMBER_APS_OPTION_ENABLE_ROUTE_DISCOVERY are both set in the message options.

3.4.4 Retries and Acknowledgements

ZigBee and its underlying network layers provide a system of retries and acknowledgments that are designed to efficiently manage the uncertainty of RF communication. It is not necessary to understand this concept in order to start using ZigBee but it may be of interest to some application developers in specific situation.

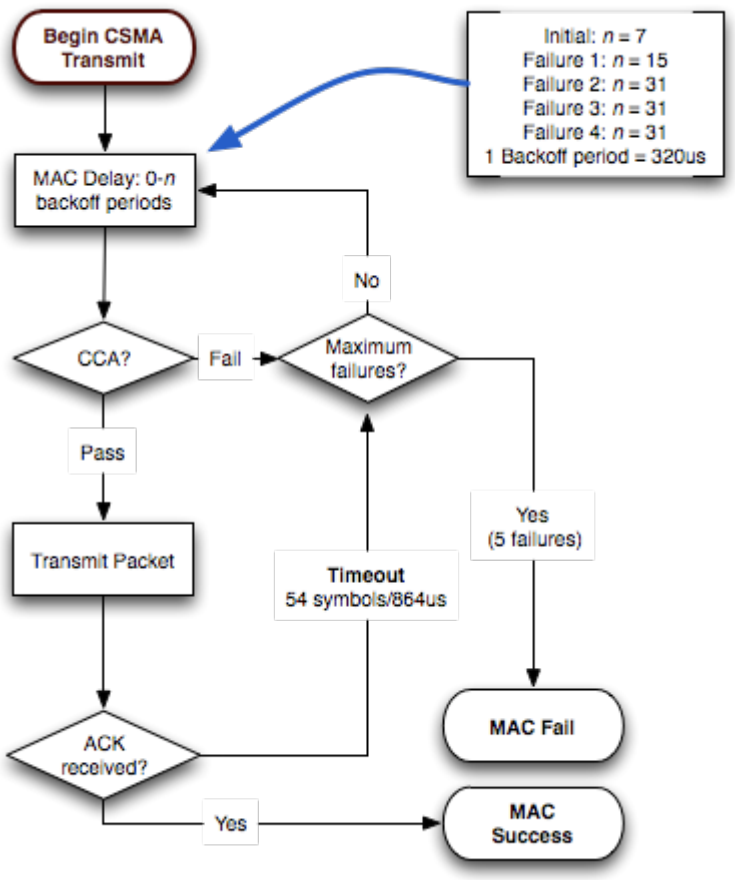
This section discusses retries and acknowledgments layer-by-layer:

- MAC retries and ACKs (802.15.4)
- NWK retries (ZigBee NWK layer)
- APS retries and ACKs (ZigBee APS layer)

3.4.4.1 MAC retries and ACKs (802.15.4)

Figure 3-9 illustrates the MAC layer transmission retry process.

Figure 3-9. MAC Retry and ACK Process



The MAC layer attempts transmission five times.

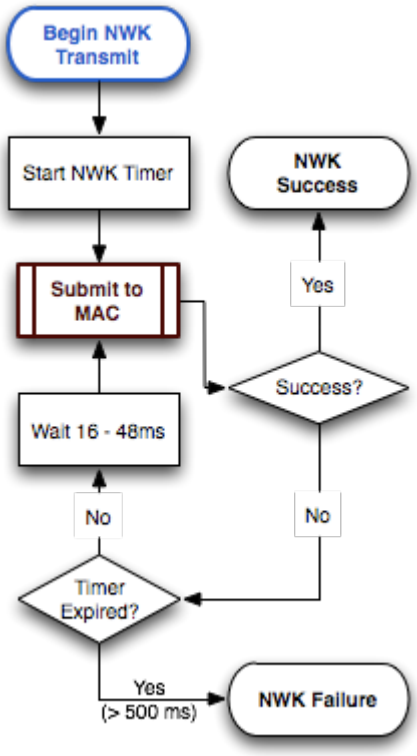
- Unicast retries will occur if the channel was not clear (CCA fail) or if the MAC level ACK was not received from the next hop destination.
- Broadcast retries will occur in the case of CCA failure but broadcast does not use the MAC ACK capability.

These retries occur very quickly — maximum retry time for complete failure is approximately 37ms. Note that the MAC ACK is sent back immediately from the sender without additional CCA — see 802.15.4 documentation for more information.

3.4.4.2 NWK retries

NWK retries in ZigBee are vendor specific. Figure 3-10 illustrates the EmberZNet NWK layer transmission retry process.

Figure 3-10. NWK RetryProcess



The NWK retries occur only if the MAC layer indicates a transmission failure. They operate on a longer time scale than MAC retries and therefore give the network additional robustness in the presence of medium term (1-500ms) interference.

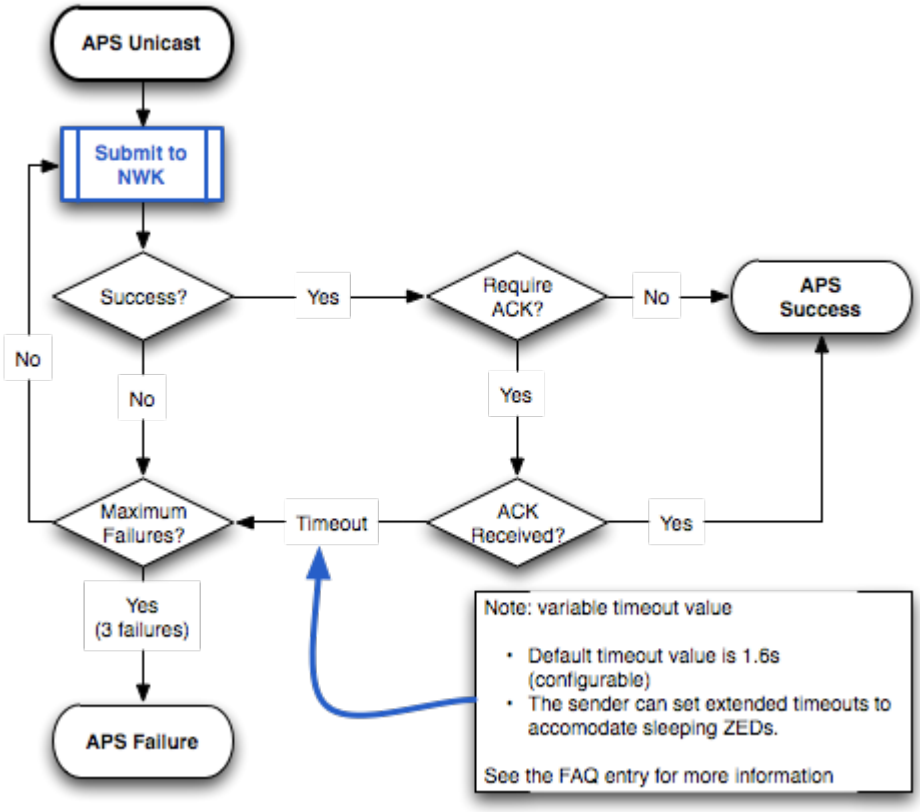
Ember's interference research shows that the NWK layer retries are important for overcoming temporary interference from WiFi in certain situations.

- Unicast: unicast behavior is as described in the flow chart, with retries for up to 500ms.
- Broadcast: broadcast messages are re-sent every 500ms up to a total of 3 times (including the initial broadcast) or until all neighbors are heard to rebroadcast the message themselves (thereby ensuring complete delivery).

3.4.4.3 APS retries and ACKs

Figure 3-11 illustrates the APS layer retry process.

Figure 3-11. APS Layer Retry Process



The APS layer has an ACK flag that controls whether it uses the additional logic to wait for an acknowledgment and retry if the acknowledgment is not heard. This represents a full end-to-end acknowledgment from the recipient device.

The APS layer can be further optionally configured to repair the route to the destination in the case that the APS sending fails.

There is no equivalent of the APS end-to-end acknowledgment for broadcast messages.

3.4.4.4 Conclusions

If it is possible to send a message to the destination, the automatic ZigBee behavior provides “maximum effort”, by attempting retries over several different time scales, from 1ms to several seconds with optional end-to-end ACK and route repair.

If the delivery fails the application is recommended to wait a more significant amount of time before retrying — this gives the interference or failure time to clear up. In cases of extreme bandwidth congestion the application retries may actually contribute to the problem.

3.5 The ZigBee Stack

ZigBee provides several separate stacks. This has been a point of confusion so the summary is as follows:

- ZigBee 2004 – was released in 2004 and supported a home control lighting profile. This stack was never extensively deployed with customers and is no longer supported.
- ZigBee 2006 – was released in 2006 and supports a single stack known as the ZigBee stack (explained below)
- ZigBee 2007 – was released in Q4 2007. It has two feature sets, ZigBee and ZigBee PRO.

The ZigBee and ZigBee PRO stacks are complete implementations of the MAC, networking layer, security services and the application framework. Devices implementing ZigBee and ZigBee PRO can interoperate by acting as end devices in the other type of network. For example, if a network is formed around a ZigBee PRO coordinator it can have ZigBee PRO routers only, but both ZigBee and ZigBee PRO end devices.

Note: The ZigBee 2004 stack is not interoperable with ZigBee 2007.

The ZigBee stack is formed around a central coordinator and uses tree addressing to establish the network. Tree routing is normally used (although table based routing can also be used). This stack supports residential security only.

The ZigBee PRO stack is formed around a central coordinator but uses a stochastic addressing scheme to avoid limitations with the tree. Table routing is always used and additional features are available such as:

- Network level multicasts
- Many-to-one and source routing
- Frequency agility
- Asymmetric link handling
- PAN ID Conflict
- Fragmentation
- Standard or high security.

Ember does not recommend deploying systems on the ZigBee stack because the ZigBee PRO stack has a number of features that are necessary for long term network stability and reliability. The ZigBee stack is typically used for small static networks.

Note the ZigBee 2007 specification includes updates to the ZigBee stack to allow the use of frequency agility and fragmentation on this stack. This stack remains interoperable with the ZigBee 2006 stack and therefore use of these features must be limited to those networks which can handle the complexity of some devices having these capabilities and some devices not.

3.5.1 ZigBee Device Object (ZDO)

The ZDO (occupying Endpoint 0 of each node) is a stack-level entity defined by the ZigBee Networking Specification for use in network management and information

gathering. This section explains how to make use of ZDO functionality in the EmberZNet stack and/or EZSP. An example is described as well.

The ZDO, an entity in the stack, provides network management capabilities that nodes can use to learn about each other, about the network in general, or for managing certain stack-level functionality. Many features of the ZDO must be implemented as part of a ZigBee Certified Platform (ZCP) and are therefore available on all ZigBee/ZigBee PRO devices, making them useful as an interoperable way to gather and manage system properties on a ZigBee network. The ZDO is implemented through an over-the-air messaging format known as the ZigBee Device Profile. The following basic guidelines apply to using the ZDO for network management:

- Broadcasts requests must go to broadcast mask 0xFFFD.
- Requests should always be sent to destination endpoint 0, with application profile ID 0x0000, as this indicates that the ZDO (intrinsic in the stack) should handle the message. The source endpoint generally corresponds to the portion of your application that is interested in the requested information.
- Responses to ZDO requests come from source endpoint 0, to the destination endpoint corresponding to the source used for the Request.
- If a particular ZDO request handler is not implemented in the stack, a status of EMBER_ZDP_NOT_SUPPORTED (0x84) will be returned in the response.
- Cluster ID is used to indicate the Request or Response type being sent.
- Requests can be sent as unicast, broadcast or multicast, just like any other APS frame (with optional APS acknowledgement), and responses arrive in the IncomingMessageHandler like any other incoming APS message. However, certain kinds of requests, if sent as broadcasts, will not generate a response (in the interest of network bandwidth conservation).
- "stack/include/zigbee-device-stack.h" details the frame formats for commands and responses, including which request types are supported by the EmberZNet stack. Further details about the frame formats are described in the ZigBee Device Profile (ZDP) section of the ZigBee Networking Specification (ZigBee document #053474).
- Utility functions (provided as source code) for generating ZDO requests and parsing certain kinds of responses can be found in "app/util/zigbee-framework/zigbee-device-library.h". (There is also an alternative version of this file for EZSP host applications.)
- All ZDO requests and responses begin with the transaction sequence number as the first byte of the payload. This allows a request to be matched up with a particular response.

As an example, take the case of a Permit Joining ZDO transaction, which facilitates remote control of the PermitAssociation flag for one or more devices in the network.

For this particular transaction, zigbee-device-stack.h explains the following:

```
1.    // Request: [duration:1] [permit:1]
2.    // Response: [status:1]
3.    #define PERMIT_JOINING_REQUEST 0x0036
4.    #define PERMIT_JOINING_RESPONSE 0x8036
```

So a Permit Joining request frame would use cluster ID 0x0036, and the response would arrive with cluster ID 0x8036. For the request parameters, the “duration” byte corresponds to the argument you would ordinarily provide to the PermitJoining call in EmberZNet/EZSP (a number of seconds to allow joining, or 0x00 for always-off joining,

or 0xFF for always-on joining). The “permit authentication” byte is only applicable to when the recipient of the request is the Trust Center (TC) and controls whether the TC should perform authentication or "certain devices" during future association requests.

However, as the current ZigBee Networking Specification doesn’t provide any guidelines about which "certain devices" we can trust or must validate, the EmberZNet stack simply ignores the authentication parameter in this case, so this argument in the Permit Joining Request is not significant.

Some examples (current as of EmberZNet 3.0.1) of sending this request frame through the EZSP Training-Host application follow. (0x42 is used as an arbitrary value here for a sequence number; the application would substitute its own values in these places.)

Unicast to node 0x401C asking it to permit joining for the next 60 (0x3C) seconds:

```
sendunicast 0 0x401C 0x0000 0x0036 1 0 0x1140 0xFFFF 0x42 0x42 4 {42 3C
00}
```

This generates an APS ACK frame (because the RETRY bit is set in the APS options above) as well as an incoming unicast (Permit Joining Response) from the request’s destination signifying the result of the request:

```
incomingMessage
type:UNICAST profileId:0x0000 clusterId:0x8036 sourceEndpoint:0
destinationEndpoint:0 options:RETRY | ENABLE_ROUTE_DISCOVERY
groupId:0x0000 sequence:10 lastHopLqi:255 lastHopRssi:0 sender:0x401C
bindingIndex:255 addressIndex:255 messageLength:2 messageContents: 42
00
```

Broadcast to entire network asking it to permit joining forever (0xFF):

```
sendbroadcast 0xfffd 0x0000 0x0036 1 0 0x0000 0xffff 0x42 10 0x42 3 {42
FF 00}
```

No response is generated as a result (since there could be potentially many responders), but devices can now join at any router in the network after this broadcast request as been processed.

3.5.1.1 Advanced ZDO usage

Additionally, as of EmberZNet release 3.3.2, Ember has added the ability for the application (host MCU) to be informed about ZDO requests that the stack is handling, and/or informed about those that the stack does not handle (so that it can handle them itself), and also to give the host a chance to handle endpoint requests (instead of handling these automatically) if desired. Please see comments in the hal/ember-configuration.c file (in SoC platforms) or app/util/ezsp/ezsp-enum.h (for NCP platforms) in EmberZNet releases 3.3.2 and later for descriptions of these available options, which are described as EzspConfigId values 0x26-0x28 (in EZSP) or the following configuration defines (in SoC API):

```
EMBER_APPLICATION_RECEIVES_SUPPORTED_ZDO_REQUESTS
EMBER_APPLICATION_HANDLES_UNSUPPORTED_ZDO_REQUESTS
EMBER_APPLICATION_HANDLES_ENDPOINT_ZDO_REQUESTS
EMBER_APPLICATION_HANDLES_BINDING_ZDO_REQUESTS
```

3.5.2 ZigBee Profiles

On top of the basic ZigBee stack are application profiles, or simply profiles. These are developed to specify the over-the-air messages required for device interoperability. A given application profile can be certified on either the ZigBee or ZigBee PRO stack.

The existing application profile groups are as follows:

- Home Automation (HA) – Devices for typical residential and small commercial installations.
- Smart Energy (SE) – For utility meter reading and interaction with household devices.
- Commercial Building Automation (CBA) – Devices for large commercial buildings and networks.
- Telecom Application (TA) – Wireless applications within the telecom area.
- Health Care (HC) – Monitoring of personal health in the home or hospital environment.
- Retail – Monitoring and information delivery in a retail environment.
- ZigBee Light Link – Wireless control of LED lighting systems.

A ZigBee Cluster Library (ZCL) forms a generic basis for some of these application profiles. This library defines common elements that are shared such as data types and allows reuse of simple devices such as on/off switch protocols between different profiles.

Application profiles define the roles and functions of devices in a ZigBee network. Two types of application profiles are administered by the Alliance:

- Public Application Profiles are developed by members of the ZigBee Alliance so that devices from different manufacturers can interoperate.
- Manufacturer Specific Application Profiles are developed by product developers creating private networks for their own applications where interoperability is not required.

If you develop a product based upon your own private application profile, the ZigBee Alliance requires you to make use of a unique private profile identifier to ensure the product can successfully co-exist with other products. The ZigBee Alliance issues these unique private profile IDs to member companies upon request and at no charge.

An application can also be developed using a public profile with private extensions for specific features from a manufacturer.

3.5.3 ZigBee Addressing Schemes

ZigBee contains two network addressing schemes in its protocol definition. It also has the ability to assign addresses manually from a commissioning tool.

The ZigBee stack (in contrast to the ZigBee PRO stack) uses tree addressing. Under this addressing mechanism the coordinator starts the network and initiates the network

space. The number of routers and end devices are set upon start up of the network. Network addresses are assigned using a distributed addressing scheme that is designed to provide every potential parent with a finite sub-block of network addresses. These addresses are unique within a particular network and are given by a parent to its children. The ZigBee coordinator determines the maximum number of children any device within its network is allowed, and many children are allocated between router children and end devices. Each child address is related to its parents address. Every device has an associated depth which indicates the minimum number of hops a transmitted frame must travel, using only parent-child links, to reach the ZigBee coordinator. The ZigBee coordinator itself has a depth of 0, while its children have a depth of 1. Multi-hop networks have a maximum depth that is greater than 1.

The tree addressing mechanism used under the ZigBee stack has some known limitations. These include potentially running out of addresses in a local area because of the tree. Under this situation new devices cannot join the network because no suitable parent exists. The tree routing also requires rebuilding of the network, including devices receiving new addresses, to reestablish a broken parent-child link. Because of these limitations, an alternate mechanism was developed for larger networks.

The ZigBee PRO stack uses a stochastic address assignment mechanism. Under this mechanism the coordinator is still used to start the network. Each device (routers and end devices) that joins the network is given a randomly assigned address from the device it is joining. The device conducts conflict detection on this address using network level messages to ensure the address is unique. This address is then retained by a device, even if the parent address changes.

Under ZigBee PRO, provisions are intended for a commissioning tool for setup and configuration of networks. This commissioning tool can be used to provide addresses manually.

3.5.4 Extended PAN IDs

Developers who have used stack software earlier than ZigBee 2007 should note a new and important change to PAN IDs. ZigBee has added an 8 byte extended PAN ID (EPID or XPID) to facilitate provisioning and PAN ID conflict detection. The extended PAN ID is now included in the beacon payload, following the existing 3 bytes.

The Extended PAN ID is a network parameter that is new for ZigBee 2007 (EmberZNet 3.x and later) and is used in the ZigBee and ZigBee PRO feature sets. This extended PAN ID [EPID] can be thought of as an extension of the basic, 16-bit PAN ID [PID]. The EPID is a 64-bit value set for the entire network by the ZigBee Coordinator [ZC] at the time the personal area network [PAN] is formed and must not change while the PAN is operating (unlike the PID). Like the PID, all nodes within the same PAN share an EPID.

Other than the scanning and joining processes, the EPID rarely appears in transmitted ZigBee packets due to its large overhead (8 bytes) in the header. However, the EPID is used during a Network Update transaction, where a Network Manager node informs the other devices in the PAN about a PID conflict or channel change, so that nodes moving to a new channel and/or PID can rely on the consistency of the EPID as a criteria for finding the moved network.

Note that the Extended PAN ID (even if it's zero) is saved to the non-volatile memory (as part of the tokens) once the node associates into the network, so it won't change over the lifetime of the device until the device leaves the network.

Use in Scanning / Forming / Joining Process

The EPID will appear (along with other information such as the PID) in the beacon results seen by an Active Scan, and when a join is initiated, if the EPID is non-zero in the join request, the EPID will take precedence as the primary criteria for matching up the joiner to the joinee (16-bit PAN ID is effectively ignored). However, it's possible that some networks (particularly older ones) may not want (or be able) to deal with EPIDs, so the EPID would be zero in the network parameters from the beacon. If the requested EPID is all zeroes during the JoinNetwork operation, only the 16-bit PAN ID will be used to match to a network for the purposes of association.

While Ember provides some sample utilities (in /app/util/scan) for scanning networks and parsing the results to make a determination about which one to join, this process will generally vary depending on the application design and the level of strictness desired in selecting a network. (For example, the joining device may simply want any available ZigBee Pro network; it may seek a closed network designed specifically for the application in use but may not care which closed network it chooses; or it may seek one specific closed network with specific properties, in the way that I want my laptop at home to join not just any home WiFi network, but my WiFi network in my home.)

The following guidelines apply to EmberZNet in determining the expected behavior of a Form/Join action given a particular EPID:

- If an all-zero value is specified for extended PAN ID during FormNetwork, the stack will generate a random 64-bit value for this field.
- If an all-zero value is specified for extended PAN ID during JoinNetwork, the stack will use the 16-bit PAN ID specified in the JoinNetwork parameters as the primary criteria for selecting a network during the join process.
- If a non-zero value is specified for extended PAN ID during JoinNetwork, the stack will use the 64-bit extended PAN ID specified in the JoinNetwork parameters (even if 16-bit PAN ID is different) as the primary criteria for selecting a network during the join process.

Choosing an EPID

While the PAN ID is meant to be a randomly chosen, 16-bit value, unique to each network, the EPID is often used more like the SSID field of WiFi networks to give them a user-friendly designation (which is often not so random and is either set by the manufacturer or the installer). However, Ember discourages using a fixed EPID for all deployments because (unlike the conflicts of the PAN ID) EPID conflicts cannot be resolved if they occur at runtime (because no other unique information exists to distinguish the PANs). Customers wishing to use non-random EPIDs should, at minimum, scan the network (either at the coordinator or through some external commissioning tool) and check that the new PAN's preferred EPID is not already in use by some other PAN. One approach is to use a small set of preferred EPIDs when forming PANs so that the coordinator can have alternatives if the first choice is in conflict.

OEMs creating consumer-grade products requiring customer installation (rather than by trained installers) should take particular care to ensure variety of EPIDs, as two customers living next door to each other may purchase the same product for their homes and would prefer to isolate their networks from each other. If those two neighboring homes were to each use PANs with the same EPID, network difficulties would likely arise for both users because the both homes would be considered as one network, and many network address conflicts could occur.

EPID versus PID

Here is a quick overview of differences between the Exteneded PAN ID and the standard PAN ID:

- EPID is 64 bits; PID is 16 bits
- EPID is usually used as stack's criteria for matching to the requested network; PID is only used for matching criteria when EPID is all 0x00 bytes.
- EPID is only present in a few kinds of packets (beacons, Network Update messages); PID is present in almost all 802.15.4 frames (except MAC ACKs).
- EPID is used as criteria for uniquely identifying the network and for resolving conflicts of PID; PID is used for MAC destination filtering at the radio receiver.
- EPID may help provide some indication of the network's identity in the scan results; PID should always be completely random, so it is not as useful in determining which PAN is the "right one".
- EPID can be any value in range of 0x0000000000000001 to 0xfffffffffffffffe (all 0's and all F's are reserved values); PID can be any value in the range of 0x0000 - 0xFFFE (all F's constitute reserved value).

3.6 ZigBee Cluster Library

3.6.1 Overview

In the ZigBee Cluster Library (ZCL), a cluster is a set of messages used to send and receive related commands and data over a ZigBee network. For example, a temperature cluster would contain all the necessary over-the-air messages required to send and receive information about temperature.

To facilitate learning and management, these clusters are further grouped into functional domains, such as those useful for HVAC, Security, Lighting, and so on. Developers may also define their own clusters, if the pre-defined clusters do not meet their specific application needs.

ZigBee application profiles then reference which clusters are used within the profile, and specify which clusters are supported by each device defined within the profile – some clusters are mandatory, others optional. In this way, the ZCL simplifies the documentation of a particular profile and allows the developer to understand quickly which behaviors each device supports.

A more detailed overview of the ZCL, the format of messages within clusters, and a set of messages that may be used within any cluster are described in the ZigBee Cluster Library Specification document (075123r02ZB_AFG-ZigBee_Cluster_Library_Specification.pdf). Functional domain clusters are described in separate documents, such as the Functional Domain: Generic, Security and Safety document.

Ember provides source code to easily assemble and disassemble ZCL messages, whether they are pre-defined by the ZCL or custom messages created by the developer. In addition, Ember provides a powerful tool called AppBuilder, part of the InSight Desktop tool, which allows developers to create and configure most of their ZCL-based application. See Ember document 120-3038-000, *Application Framework V2 Developer Guide*, for more information.

3.6.2 Inside Clusters

3.6.2.1 Clients and Servers

Each cluster is divided into two ends, a client end and a server end. The client end of a cluster sends messages that may be received by the server end. The client end may also receive messages that are sent by the server end. In this sense, the client and server ends of a cluster are always complementary. In contrast to many other systems (for example, HTTP), both have the same potential for sending and receiving messages: the "client" designation does not imply a subordinate or response-only status.

Because all commands have a sender and a receiver, each cluster is described in two parts – a server part and a client part, as shown in Figure 3-12. A device supporting the server half of a cluster will communicate with a device supporting the client half of the same cluster.

Figure 3-12. A ZigBee Cluster

	Server	Client
Attributes	On/Off	None
Received Commands	On Off Toggle	None
Generated Commands	None	On Off Toggle

This equality complicates discussions; for clarity, this document always refers to "cluster end" when one of the client or the server end must be used, "cluster ends" when speaking of both client and server ends, and "cluster server" or "cluster client" when a specific end is required (usually examples).

3.6.2.2 Attributes

An attribute is data associated a cluster end; the server and client ends of a cluster may each possess multiple attributes.

Each attribute declares a 16-bit identifier, a data type, a read-only or read/write designator, a default value, and an indicator of whether its support by any implementation is mandatory or optional. Table 3-1 lists the most common data types. The data types are fully described in The ZCL Specification. Table 3-2 is an excerpt from that document.

Table 3-1. Data type quick reference (most common data types)

Data Type	Description
Binary data types:	8, 16, 24, or 32 bits in length
Logical data type:	Boolean
Bitmap data type:	8, 16, 24, or 32 bits in length
Unsigned Integer:	8, 16, 24, or 32 bits in length
Signed Integer:	8, 16, 24, or 32 bits in length
Enumeration:	8 or 16 bits in length
Floating Point:	2-byte semi-precision, 4-byte single precision, or 8 byte double precision
String:	binary octet string or character string; first byte is length
Time:	time of day, date
Identifier:	cluster ID, attribute ID
IEEE address type	

The attribute identifier is unique only within the specific cluster end: this means that the attribute 0x0002 within the cluster server does not need to be the same as the attribute 0x0002 within the cluster client, even within the same cluster.

Table 3-2. ZCL Data Types

Type Class	Data Type ID	Data Type	Length of Data (octets)	Invalid Number	Analog / Discrete
Null	0x00	No data	0	-	-
	0x01 – 0x7	Reserved	-	-	
General Data	0x08	8-bit data	1	-	D
	0x09	16-bit data	2	-	
	0x0a	24-bit data	3	-	
	0x0b	32-bit data	4	-	
	0x0c – 0x0f	Reserved	-	-	
Logical	0x10	Boolean	1	0xff	D
	0x11 – 0x17	Reserved	-	-	
Bitmap	0x18	8-bit bitmap	1	-	D
	0x19	16-bit bitmap	2	-	
	0x1a	24-bit bitmap	3	-	
	0x1b	32-bit bitmap	4	-	
	0x1c – 0x1f	Reserved	-	-	
Unsigned Integer	0x20	Unsigned 8-bit integer	1	0xff	A
	0x21	Unsigned 16-bit integer	2	0xffff	
	0x22	Unsigned 24-bit integer	3	0xffffff	
	0x23	Unsigned 32-bit integer	4	0xffffffff	
	0x24 – 0x27	Reserved	-	-	
Signed Integer	0x28	Signed 8-bit integer	1	0x80	A
	0x29	Signed 16-bit integer	2	0x8000	
	0x2a	Signed 24-bit integer	3	0x800000	
	0x2b	Signed 32-bit integer	4	0x80000000	
	0x2c – 0x2f	Reserved	-	-	
Enumeration	0x30	8-bit enumeration	1	0xff	D
	0x31	16-bit enumeration	2	0xffff	

Type Class	Data Type ID	Data Type	Length of Data (octets)	Invalid Number	Analog / Discrete
	0x32 – 0x37	Reserved	-	-	
Floating Point	0x38	Semi-precision	2	Not a Number	A
	0x39	Single precision	4	Not a Number	
	0x3a	Double precision	8	Not a Number	
	0x3b – 0x3f	Reserved	-	-	
String	0x40	Reserved	-	-	D
	0x41	Octet string	Defined in first octet	0xff in first octet	
	0x42	Character string	Defined in first octet	0xff in first octet	
	0x43 – 0x47	Reserved	-	-	
Array	0x48 – 0x4f	Reserved	-	-	-
List	0x50 – 0x57	Reserved	-	-	-
Reserved	0x58 – 0xdf	-	-	-	-
Time	0xe0	Time of day	4	0xffffffff	A
	0xe1	Date	4	0xffffffff	
	0xe2 – 0xe7	Reserved	-	-	
Identifier	0xe8	Cluster ID	2	0xffff	D
	0xe9	Attribute ID	2	0xffff	
	0xea	BACnet OID	4	0xffffffff	
	0xeb – 0xef	Reserved	-	-	
Miscellaneous	0xf0	IEEE address	8	0xffffffffffffffff	D
	0xf1 – 0xfe	Reserved	-	-	-
Unknown	0xff	Unknown	0	-	-

Attributes may be accessed over-the-air by use of the attribute commands described later in this chapter.

3.6.2.3 Commands

A command is composed of an 8-bit command-identifier and a payload format. Like attributes, the 8-bit identifier is unique only within the specific cluster end. The payload format is arbitrary to the command type, conforming only to the general packet format guidelines as described in the ZCL Specification.

Commands are divided into two types: profile-wide and cluster-specific. Cluster-specific commands are defined inside the cluster definitions in the ZCL functional domain

documents, and are unique to the cluster in which they are defined. Profile-wide commands are defined in the ZCL Specification and are not specific to any cluster.

Profile-Wide Commands

Profile-wide commands are not unique to a specific cluster; they are defined in the ZCL General Command Frame.[2] Table 3-3 lists profile-wide commands:

Table 3-3. Profile Wide Commands

Messages Sent to the Cluster End Supporting the Attribute	Messages Sent From the Cluster End Supporting the Attribute
Read Attributes	Read Attributes Response
Write Attributes	Write Attributes Response/No Response
Write Attributes Undivided	Write Attributes Response/No Response
Configure Reporting	Configure Reporting Response
Read Reporting Configuration	Read Reporting Configuration Response
Discover Attributes	Discover Attributes Response
	Report Attributes
Default Response	Default Response

- Read Attributes: Requests one or more attributes to be returned by the recipient; replies with Read Attributes Response.
- Write Attributes: Provides new values for one or more attributes on the recipient; the reply will contain a Write Attributes Response portion indicating which attributes were successfully updated, and/or a Write Attributes No Response portion for attributes that were not successfully updated.
- Write Attributes Undivided: Updates all attributes and replies with Write Attributes Response; if any single attribute cannot be updated, no attributes are updated and this command replies with Write Attributes No Response.
- Configure Reporting: Configures a reporting interval, trigger events, and a destination for indicated attributes. Replies with Configure Reporting Response.
- Read Reporting Configuration: Generates a Read Reporting Configuration Response containing the current reporting configuration sent in reply.
- Report Attributes: A report of attribute values configured by Configure Reporting command.
- Default Response: A response sent when no more specific response is available (and the default response is not disabled by the incoming message).
- Discover Attributes: Requests all supported attributes to be sent; replies with a Discover Attributes Response.

Since attributes are always tied to a cluster, the commands affecting attributes specify which cluster and which attributes are to be accessed or modified. Additionally, each cluster defines which attributes support which commands – for example, an attributes may be declared READ ONLY, in which case it will not support the Write Attributes

[2] See Chapter 7, ZCL Specification 075123r02.

command. Thus, while the command format is not cluster-specific, the attributes it describes and its result on the receiving system are both cluster-specific.

Readers interested in more detail about the format or specific behaviors of these messages should review the ZCL Specification (075123r02).

Cluster-Specific Commands

The payload format, support requirements (mandatory, optional), and behavior on receipt of a cluster-specific command are all defined in the cluster definition. Typically, these commands affect the state of the receiving device and may alter the attributes of the cluster as a side-effect.

For example, the ZCL defines three commands (OFF, ON, and TOGGLE) that are received by the On/Off cluster server. It further declares that each of these commands is mandatory and the payload format for each command (in this case, none of them have payloads). The ZCL defines that the On/Off cluster client is responsible for generating the commands received by the server.

3.6.3 Example: Temperature Measurement Sensor Cluster

As an example, consider a portion of the Temperature Measurement Sensor cluster that is fully described in chapter 4 of the ZigBee Cluster Library Specification document (075123r02ZB_AFG-ZigBee_Cluster_Library_Specification.pdf).[3] Table 2 4 is taken directly from this document.

Table 3-4. Temperature Measurement Sensor Server Attributes

Identifier	Names	Types	Range	Access	Default	Mandatory/ Optional
0x0000	MeasuredValue	Signed 16-bit Integer	MinMeasuredValue to MaxMeasuredValue	Read only	0	M
0x0001	MinMeasuredValue	Signed 16-bit Integer	0x954b – 0x7ffe	Read only	-	M
0x0002	MaxMeasuredValue	Signed 16-bit Integer	0x954c – 0x7fff	Read only	-	M
0x0003	Tolerance	Unsigned 16-bit Integer	0x0000 – 0x0800	Read only	-	O

Step 1. Overview of Attributes

The cluster server supports four attributes, three of which must be supported by any implementation (MeasuredValue, MinMeasuredValue, MaxMeasuredValue), and one of which may be optionally supported (Tolerance). All of these clusters are read-only, indicating that any write attempts to them will fail.

[3] Note that all ZigBee document are available on the ZigBee website www.zigbee.org. Membership is required to access specification documents.

Step 2. Implications for Cluster Server, Cluster Client
Clearly, the cluster server should be implemented by the device that contains the temperature sensor. Meanwhile the cluster client should be implemented by any device that wishes to receive temperature sensor information, either actively (through a read attributes command) or passively (through first a configure report attributes command and then from report attributes).

Step 3. Further Information
The cluster description also provides useful information about the actual format of the data (for example, the range of the signed 16-bit integer for MaxMeasuredValue) and the mandatory supported operations – not all attributes support all basic commands. For example, MaxMeasuredValue is a read-only command and cannot be written.

Note: While the incoming write attribute commands are supported, in this case MaxMeasuredValue always generates a Write Attributes No Response reply.

Step 4. Commands
No custom commands are supported by this cluster (either the server or the client). For an example of a cluster containing custom commands, see the Thermostat Cluster in the ZCL.

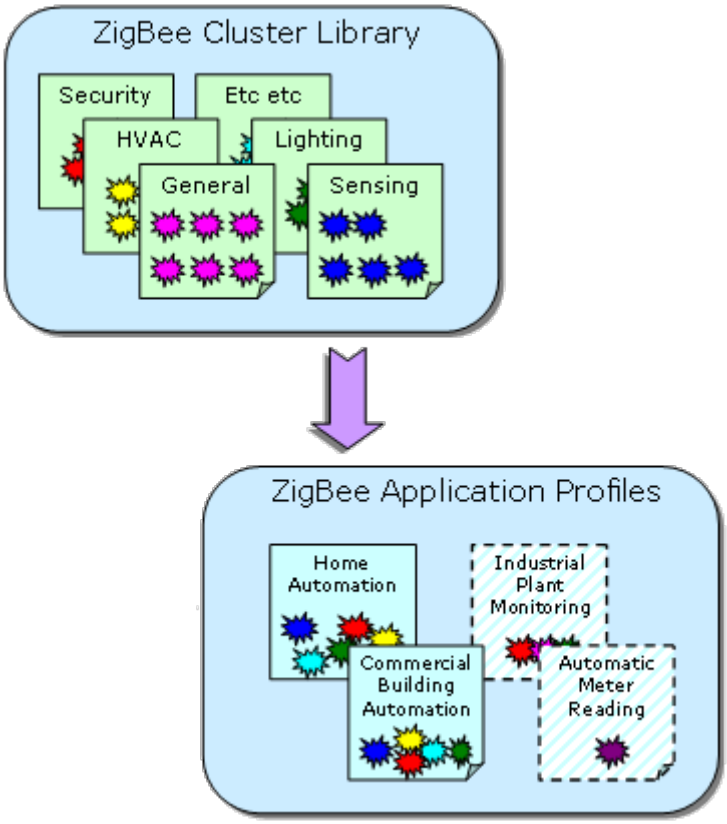
3.6.4 Functional Domains

As of this writing, the ZigBee Cluster Library defines the following functional domains:

- General
- Closures
- HVAC
- Lighting
- Measurement & Sensing
- Security & Safety
- Protocol Interfaces

Each domain defines a number of clusters that are then used by ZigBee Application Profiles to describe the over-the-air behavior of devices in the profile (see Figure 3-13).

Figure 3-13. Cluster Library Functional Domains



3.6.5 Manufacturer Extensions

The ZCL allows extension of the existing library in two ways: users may add manufacturer specific commands or attributes to existing clusters, or they may define entirely new clusters that are manufacturer specific.

Manufacturer-specific commands are identified by setting a special bit in the ZCL header and including the manufacturer code (received from the ZigBee Alliance) in the ZCL header. This guarantees that manufacturer-specific extensions do not interfere with other manufacturer-specific extensions or existing ZCL clusters, commands, or attributes.

3.7 ZigBee Compliance

ZigBee compliance is based on a building block of compliance testing used to ensure that each layer is tested.

Products that meet all compliance requirements may be branded "ZigBee Compliant" and can display the ZigBee logo.

Figure 3-14. ZigBee Logo

The ZigBee radio and MAC are required to have passed the applicable parts of IEEE 802.15.4 certification testing. Parts of the MAC not required for ZigBee operation are not required for this testing.

ZigBee stacks are required to be built on certified IEEE 802.15.4 platforms. ZigBee stack providers are required to have the stack tested against a standard ZigBee test plan known as ZigBee Compliant Platform Testing. This testing validates the basic network, security, and ZigBee Device Object (ZDO) operations for the stack.

ZigBee products are required to be built on a ZigBee Compliant Platform. The developer can choose to build a public or manufacturer-specific application. Those applications which are manufacturer-specific are tested against a specific set of tests to validate basic operation. Public profiles are more extensively tested using a standard ZigBee test for the application layer commands.

Each shipping product has to be certified and substantive changes to the product after certification can require recertification. Ember sends all chips and stack releases through compliance testing.

All of the ZigBee test plans are developed and tested using at least three members' products to ensure interoperability between different implementations.

Companies are required to become members of the ZigBee alliance to ship product containing the ZigBee stack. There are specific rules for use of the ZigBee logo on product that are detailed on the ZigBee web site (www.zigbee.org).

3.8 ZigBee IP

Under cooperative efforts with the equipment manufacturers, utilities, and United States state and federal regulators, ZigBee initiated the development of a ZigBee IP stack. The goals of this effort are to develop a ZigBee stack that provides cleaner interfaces to the existing IP infrastructure and is based on existing or developed specifications from standards organizations such as IEEE, IETF, IEC or other Internationally recognized standards bodies. This evolution of the existing ZigBee stack is underway now and will take some time to complete. For an update on the latest schedule for these efforts contact your local Ember Sales Representative or the Ember Support Team.

The ZigBee IP stack will continue to be based on IEEE 802.15.4 (although it uses 2006 rather than the 2003 version used by ZigBee PRO). It will use a new routing protocol developed by IETF. The application layer is expected to use a form of compressed XML over standard UDP and TCP mechanisms. The IETF method for using IPv6 over 802.15.4 networks known as 6lowPAN will also be used.

This ZigBee IP stack will become a requirement for the United States utility industry but is not a requirement for the other ZigBee application profiles. These profiles will make a decision on adopting this stack once it is closer to completion and the impact of its use known.

3.9 Applying ZigBee

A set of design decisions must be made in any ZigBee implementation. While these areas of application development are covered in more detail in other chapters of this guide, the initial design choices are summarized as follows:

1. Implement ZigBee at the board level or as a module – A number of ZigBee modules are now available that provide different form factors, use of power amplifiers for increased range, and module level certification to simplify the hardware integration efforts of a ZigBee design.
2. Select a ZigBee Platform – The available ZigBee platforms from Ember include the EM250, EM260, EM351, and EM357. The EM2420 is not recommended for new designs but continues to be supported for legacy designs. The EM250, EM351, and EM357 are single chip implementations designed for low cost implementations. The EM260 is a ZigBee coprocessor designed for use with a host microprocessor or for those applications requiring increased memory or specialized peripherals not available on the EM250, EM351, or EM357. The first design choice in applying ZigBee is to select which type of device is to be used. The Ember API for implementation of ZigBee is standard across these platforms to make switching applications simple. Many networks are mixed with EM250s, EM351s, and EM357s in some devices and EM260s in other devices.
3. Select a Bootloader – Several choices are available for in-the-field device upgrades. The simplest choice is to not upgrade devices once deployed (no bootloader). However, this is not recommended as issues often are found during initial field trials or beta testing that require software updates. Ember provides two commonly-used bootloaders. One is a stand-alone bootloader that provides for using and upgrading the entire flash contents. This bootloader is limited to one hop data transfer. The application bootloader can be used across multiple hops under the application control, but requires extra flash to store the image being bootloaded. ZigBee is also now developing a standardized application bootloader to provide a common mechanism for devices to be upgraded in the field. Ember is active in the development of this specification and will support this bootloader once completed.
4. Design the Data Flow and Message Types to be Used – Based on the application, the expected flow of data in the network can be determined. The use of APS level messages, multicasts, broadcasts, or many-to-one routing has to be considered.
5. Develop and Debug the Application – The design is implemented and tested in the lab and office environment. During this period, the development and debug tools are critical to evaluate the network and application. To provide for quicker debug and development, Ember recommends that InSight desktop be used with nodes connected over the InSight Adapter hardware during initial development and testing.
6. Conduct Field Trials – Once lab and office testing is completed, more extensive field trials should be conducted at typical expected installations. These installations should be carefully monitored and evaluated for improvements to the application.
7. Perform Manufacturing Tests – Development of production level hardware requires consideration of the manufacturing testing to be conducted and how the application supports these tests. Ember has an embedded manufacturing library to assist in this testing.

4 Fundamental Design Choices

This chapter describes the application design process in terms of major decisions that must be made about how to architect a solution. These design choices include:

- Whether to use Ember's Application Framework or create a custom design adapted from other sample code.
- Whether to use a system-on-chip (SoC) design or a network coprocessor (NCP) design
- How to find the right network
- How to match up devices to available services in the network
- What kind of routing optimizations to employ in the network
- How to deliver messages through the network
- What kind of security to use

Once you have considered these choices, you can begin implementing the system design.

4.1 Use Ember's Application Framework or Create an Adapted Design?

4.1.1 Application Framework-Based Design or an Adapted Design

Any application may be built from scratch, but this is a slow and sometimes tedious process. The alternative is to take a working application and modify it to meet the requirements of your application. Adapting a working design is an easier and more efficient approach to building an application, especially your first application using a new technology. Ember recommends this approach and provides tools and examples for this purpose.

As part of the InSight Desktop tool chain, Ember supplies AppBuilder and the Application Framework. AppBuilder is an interactive GUI tool that allows you to configure a body of Ember-supplied code to implement ZigBee-defined devices, such as a Smart Energy meter or a Home Automation On/Off switch, within an application profile. The Ember Application Framework is an application framework based around the ZigBee Cluster Library (ZCL). The Application Framework provides a set of embedded code to implement ZCL-based cluster handling, required network tasks, and typical ZigBee application-state machines according to best practices recommended by Ember and by the specifications for different ZigBee application profiles. This code acts as a sort of "software reference design" that can be run on either an Ember System-On-Chip (such

as the EM250, EM351, or EM357) or on a host processor connected to an Ember Network Control Processor (such as the EM260) based on the EmberZNet Serial Protocol (EZSP). The Application Framework provides common basic services and, when combined with device-specific code, provides you with most of the over-the-air behavior for your device. Your task then becomes developing hardware-specific functionality and interfacing it to the Ember software.

Additionally, Ember supplies several sample applications that are installed along with your EmberZNet PRO stack software. These sample applications are discussed in detail in the "sampleApps.html" documentation found in the "app" folder of your EmberZNet PRO installation. Some of these samples are intended to illustrate specific features or functional applications of EmberZNet PRO. If you cannot or do not wish to use Ember's Application Framework, you can adapt existing sample applications or code snippets (such as those found in the "util" folder hierarchy of the EmberZNet PRO installation) into a custom framework of your own design, without being constrained by a third-party hierarchy.

4.1.2 Application Framework Benefits

Using the Application Framework has the following benefits:

- Helps maintain a common approach, and adherence to the ZigBee standard and Ember best practices.
- Helps ensure ZigBee compliance, since Ember runs its Application Framework through pre-compliance testing
- Ember maintains the core components, allowing easy merges of updates.
- Offers the ability to include complex tasks in your application, without the need to manage those tasks at the code level.
- Provides configurability while maintaining ZigBee compliance and the ability to easily upgrade to future releases of Ember code.
- Includes a graphical user interface for setup.
- Provides a callback/command interface and set of configuration points that are farther removed from the stack than usual, and therefore are more in tune with application-layer behaviors than stack level behaviors.

All of the above help to achieve faster time to market and more robust solutions.

4.1.3 When not to use AF

The Application Framework offers little advantage in the following situations:

- The application does not need ZigBee compliance or interoperability.
- The application is for a function or use case well outside of normal ZigBee design (such as non-ZigBee messaging).
- The applications are not stack-based (such as drivers, HAL testing, or token utilities).

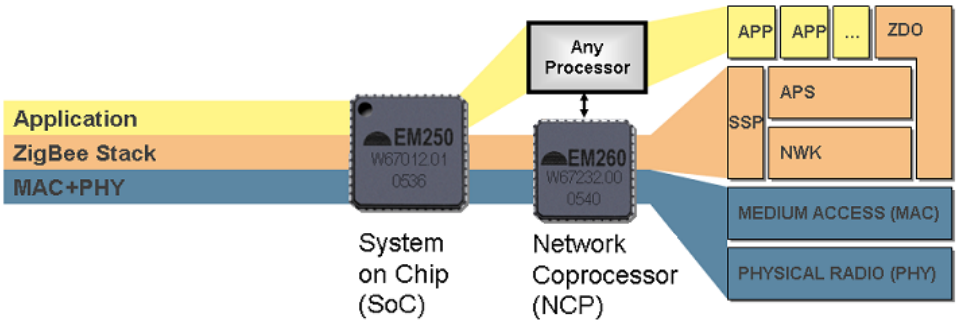
4.2 SoC or NCP?

Regardless of whether Application Framework is used for the design, the choice of the design paradigm – either the system-on-chip (SoC) model or the serial network coprocessor (NCP) model – is a crucial one. It dictates the requirements and constraints of both the software and the hardware. This choice governs where the application resides relative to the core stack functionality. In the SoC model the entire system

(stack and application) resides on a single chip, whereas in the NCP model the stack processing is done in a separate "coprocessor" that interacts with the application's own microcontroller through an external serial interface.

Figure 4-1 illustrates the various components of the stack and application and how they are organized relative to the SoC or NCP architecture model.

Figure 4-1. Stack and Application Component Organization in SoC and NCP Architectures



Note: While the choice between architectures is not to be made lightly, the Application Framework masks the differences to some degree, simplifying a change from one architecture to another when necessary, or supporting a mix of architectures for different products.

4.2.1 System-on-Chip Approach Using EmberZNet PRO API

In the SoC approach, a single chip, such as Ember's EM250 or EM357, provides all stack functionality (including integrated flash, RAM, and RF transceiver) as well as the application-layer components (application profiles, clusters, attribute management, and stack interactions). Stack functionality is implemented as pre-compiled library files, which you then must link with, along with your own application-related code, during the final build process to produce a single, monolithic binary image comprising everything needed for a completely functional ZigBee application. The Application Framework, although supplied by Ember, is considered part of the Application Layer.

Note: While a bootloader is typically used in deployed ZigBee devices, that bootloader firmware is not part of this monolithic binary image. However, Ember does provide post-build tools that can be used to further combine both the application firmware and the stack firmware into a single HEX record file for ease of distribution and manufacturing. For more information about these software utilities, please consult Ember documents 120-4020-000, *EM2xx Utilities Guide,* and 120-4032-000, *EM3xx Utilities Guide,* for the EM200 series and EM300 series platforms, respectively. More information about the bootloader can also be found in Chapter 7: Bootloading.

In the SoC approach to development, the application, including the Application Framework, is co-resident with the EmberZNet PRO stack. The application calls application programming interface functions (APIs) provided by the EmberZNet PRO stack libraries, and the stack triggers handler functions implemented by the application code. When the Application Framework is used for the application design, the framework handles calling these APIs and implementing the necessary handler

functions, then wraps these in higher level APIs and application callbacks to simplify the design process and help ensure ZigBee compliance.

Because the SoC model requires only a single chip, compared to the NCP model and legacy design architectures that require multiple ICs, the SoC model has lower power consumption, lower bill of materials (BOM) cost, and smaller possible layouts. Also, tighter integration with the stack software and radio hardware can be achieved when everything resides on a single chip, allowing for more precise and timely control over application behaviors correlated to stack activity.

However, once you have committed to an SoC model, you are bound by the constraints of the available offerings in that SoC family. These include:

- Flash and RAM memory constraints
- Toolchain constraints, such as the requirement to use xIDE for the EM250 SoC or IAR Embedded Workbench for the EM35x SoCs
- HAL constraints, such as limited amount of peripherals of a certain type, or lack of a specialized peripheral that may be integral to your hardware design
- Timing constraints based on having to share a CPU with the stack, which has its own set of requirements in order to maintain IEEE 802.15.4 and ZigBee compliance

If any of these constraints are too much of a deterrent, the NCP model may be a more attractive alternative.

4.2.2 NCP Approach with EZSP

In the NCP approach, an Ember chip with integrated flash, RAM, and RF transceiver runs most stack functions on its own through pre-loaded coprocessor firmware with runtime configurability, then uses a serial interface such as SPI or UART to communicate with a second device, known as the "host" processor, on which the application layer functionality is "hosted" separately from the core stack components. The NCP may be a special integrated circuit designed with limited I/O and reduced functionality for the express purpose of acting as a coprocessor, such as Ember's EM260, or it may be a fully-featured microcontroller (like the EM357) that happens to have the coprocessor firmware loaded onto it to make it behave as an NCP.

To facilitate communication between the application's host and the stack's NCP, Ember provides a serial command set known as EZSP, the EmberZNet Serial Protocol. (See Ember document 120-3009-000, *EZSP Reference Guide*, for more information about EZSP.) This protocol, which operates either synchronously over SPI or asynchronously over UART (with or without flow control), mimics the EmberZNet PRO API with EZSP command frames and the EmberZNet PRO handler functions with callback response frames. Ember provides EZSP driver source code that abstracts these serial commands and responses into a set of APIs and handler functions similar to those used in the SoC model. When Ember's Application Framework is used to implement the application layer, it takes care of calling the necessary API functions and implementing the required handler functions, allowing the designer to focus on higher level application processing with client APIs and framework callbacks.

The main advantage to the NCP platform is its flexibility. The host processor can be as simple as an 8-bit microcontroller with 16 KB of flash and 4 KB of RAM, or it can be something as sophisticated as a 64-bit computer with gigabytes of memory and a Windows or Linux operating system. This means that the NCP design is well-suited to scenarios where ZigBee is being added on or retrofitted to an existing system, such that an OEM's expertise and intellectual property on the software and hardware side can be leveraged to speed up the design cycle and expedite time to market. Another advantage

to the NCP approach is that the host can provide significantly more resources (flash and RAM) and a different peripheral set for the application than the available SoC options. This allows for development of more complex applications with new features, and protects the application from exceeding the limitations of the SoC when significant new features are added to the application.

Decoupling the stack processing from the application allows installing fixes and new features on the stack side with simple firmware updates to the NCP, without necessarily requiring any changes to the application firmware on the host. This decoupling also removes the CPU time constraints of sharing a processor with the stack. Because the NCP firmware manages the sleep state of the NCP to minimize its activity and its current consumption, only the host processor needs to be active when the application has tasks that don't directly involve the stack. If the host processor's active current consumption is lower than that of the SoC when the CPU is active without the radio, the total current consumed by the host and the NCP when running non-networking application tasks may actually be lower than the SoC in a comparable scenario.

The primary disadvantage to the NCP approach is the addition of a second, host, processor, which adds extra cost and PCB real estate, and may impact the device's overall power consumption. Another tradeoff is that decoupling stack and application processing means that certain time-sensitive interactions between the stack and the application can no longer occur in "real time" and must instead occur as notifications about decisions made by the stack after the fact. Thus, the host application has fewer opportunities to decide the outcome of certain decisions as they arise. Instead, "policies" are configured on the NCP to guide stack behavior in those situations. Also, because the NCP firmware is pre-built firmware supplied by Ember, the application designer does forfeit some amount of control over how the stack behaves and how its internal resources are allocated.

Once you commit to using the NCP approach with EZSP, you must then decide which host platform to use for the design. This platform may be different for the prototyping and final design stages, depending on the availability of materials and the flexibility required during the initial stages of debugging. When choosing a host platform, consider your existing expertise and available tools and resources on that platform, the cost and power consumption requirements of that platform, and the amount of memory available for application development, including any headroom needed for future enhancements. You should also consider whether to use UART or SPI for EZSP communication. EZSP-UART requires a more complex driver, generally intended for use in a POSIX-compliant operating system, with more sophisticated logic and a larger memory footprint than the EZSP-SPI driver, and its supported maximum throughput is not as high. However, the EZSP-SPI implementation requires a few more interface pins than the EZSP-UART design. Because not all microcontrollers or operating systems support SPI, architectural constraints at the host may dictate this design choice.

4.2.3 Differences in Design

Table 4.1 shows some of the primary differences between an SoC application and an NCP-based host application, by function.

Table 4-1. Functional Differences, SoC compared with NCP

Method	Difference
Managing stack parameters, such as table sizes and allocation limits, and endpoint descriptor data, such as support clusters and profiles	
SoC	Mainly set up through compile-time definitions built statically into the application binary.
NCP	Managed by the NCP but configured by host at runtime after boot-up of NCP and before engaging in any network activity; this interval is referred to as the “Configuration” phase and allows dynamic configuration of the NCP without rebuilding its firmware.
Application reaction to events	
SoC	The application can react to events, such as security authentication request, an incoming data poll from a child, or a remote binding modification, in the moment and can handle events on a case-by-case basis.
NCP	The host application configures policies ahead of time to pre-determine the outcome; notifications are after the fact.
Polling (for sleepy end devices that need to poll the network periodically) **The Application Framework takes care of the polling state machine, so the difference is negligible when it is used.**	
SoC	The application controls when each poll occurs and chooses how to react to the result of each poll.
NCP	The host application configures poll rate and failure tolerance. NCP handles polling with the configured rate, and only notifies the host when the failure rate surpasses the configured threshold. This can make the application state machine design easier on EZSP host platforms if not using the Application Framework.
Managing message buffers **The Application Framework handles SoC buffer management, so the difference is negligible when it is used.**	
SoC	The application shares memory for packet data with the stack. Shared message buffers must be allocated by the application for outgoing message data, and by the stack for incoming or relayed message data. Buffer management process, including buffer allocation/deallocation and construction, can be tricky and is often the source of errors in SoC application design.

Method	Difference
NCP	NCP handles the buffer management itself and accepts / delivers message payload data as a simple array and a length argument. This makes messaging interface simpler and less error-prone if not using the Application Framework.
Handling source routing, where the application is expected to handle incoming Route Record notifications and preserve this data in a "source route table" (see Chapter 3, section 3.4, ZigBee Routing Concepts.))	
SoC	Source route table resides on SoC; its size is constrained by the SoC's limited RAM resources.
NCP	NCP can collect the last N source routes in its own internal source route table, where N is sized by the host application during configuration. However, the host receives route record data in callbacks from the NCP and can generally buffer much more route data than the NCP, depending on the host's RAM constraints. This makes the High RAM Concentrator operation (often used for gateways, commissioning/configuration tools, and other major aggregation points) more feasible on an NCP platform than on an SoC platform, especially those SoC platforms where RAM availability is limited.

4.3 Common design themes

Although the Application Framework simplifies and abstracts the design process, some design decisions must be made as part of implementation regardless of whether the design is based on Ember's Application Framework or not. The following sections cover areas you should consider in designing your system.

4.3.1 Network Discovery / Commissioning

Commissioning refers to the process of getting devices into the network. Recall from discussion of the network joining process in Chapter 3 that, unless a device is acting as coordinator for a PAN, it must request to join an existing network, and that the joining device must scan one or more channels to locate the available networks. However, as the network coordinator has several radio channels from which to choose in forming its PAN, and since the network's PAN ID and Extended PAN are often randomized, your application generally requires some intelligence or external mechanism to assist with network discovery and commissioning. Tasks include helping ensure that the device can either join the proper network or receive the desired network settings from some external source, and ensuring that the device can be removed from the network when either the wrong network is joined by mistake or the device is being migrated to a new installation. Likewise, if you are designing a device that may act as a ZigBee PAN coordinator, it is important to consider ways in which you can ease the process of network selection for devices looking to enter your coordinator's network.

Note: If you are designing an application for use in an official, public ZigBee application profile (such as Home Automation), Ember recommends that you review the latest published revision of the appropriate ZigBee application profile specification (as

obtained from http://www.zigbee.org) for your target design, to ensure that it meets any profile-specific requirements or best practices for commissioning.

4.3.1.1 Simplifying Network Selection through Extended PAN ID or Channel Mask

Although Extended PAN ID selection by the PAN's coordinator is generally random, a proprietary network deployment may use a specific bitmask of extended PAN IDs as a way to enhance network selection for joining devices. In this model, the coordinator forms a network within this agreed-upon Extended PAN ID mask, such that joining devices could scan channels for open PANs and limit those outside of the configured Extended PAN ID range. However, this method of enhancing network selection is not feasible for devices wishing to interoperate on public ZigBee profiles with devices from a wide range of manufacturers. Because the public ZigBee application profiles do not generally limit their Extended PAN ID selection, another vendor's device may occupy Extended PAN IDs outside of the limiting bitmask that you've chosen.

Similarly, although 2.4 GHz ZigBee networks can occupy any of 16 different channels, the joining device may be able to limit its mask of channels to scan. The expected network might be a proprietary design in which the coordinator has chosen to confine its channel selection to only a few channels within a preconfigured mask. Alternatively, the application profile upon which one or more endpoints of the included devices is based may require constraining the network's channel selection to a specific set of channels. For instance, both the SE and HA application profiles require that preference be given when forming a network to channels outside of the most commonly used Wi-Fi channel allocations (channels 1, 6 and 11 in the IEEE802.11 range), which allows the joining device to confine its channel scan to ZigBee channels 11, 14, 15, 19, 20, 24, and 25. Note that Ember's Application Framework globally defines the `EMBER_AF_CHANNEL_MASK` symbol, which can be used to supply the channel mask for the device when joining or forming a network. This value is configured in your AppBuilder-generated configuration header file, as the choice of a public profile will impact which channels are recommended.

4.3.1.2 Permit Joining Control

Devices looking to join a network generally only consider those PANs that are open to new devices (in other words, they permit joining), and devices must not leave their `permitJoining` flag permanently set, at the risk of failing ZigBee compliance testing for public profiles and manufacturer-specific profiles (MSPs). Therefore, devices, especially the PAN coordinator, must ensure that they can enable the `permitJoining` flag locally for at least some limited time when new devices need to be added to the network. This enabling generally must come from some external stimulus, which will depend on the physical capabilities of a device. If a button or serial interface is available to the device, this is usually an appropriate stimulus to enable `permitJoining`. However, if the device doesn't have an external input to act as this stimulus, other methods must be considered. One possibility is to have the device enable `permitJoining` for a limited time when it is first powered on. Another option is to cause `permitJoining` to be enabled when a particular message is received by the node over the air.

With regard to the latter method, while it is possible for the application to make a local change to its own `permitJoining` state through a local call to the `emberPermitJoining()` API or `permitJoining` EZSP command, it is also possible to send a standard request through the ZDO, which is implemented intrinsically by the stack, to a ZigBee node to ask it to change its `permitJoining` state. When a ZDO Permit Joining Request is received over the air for endpoint 0 (the ZDO) on application profile 0x0000 (the ZigBee Device Profile), the stack automatically alters the

permitJoining state on the device. A unicast or broadcast of this request provides a standard way to change the joining permissions of the network remotely for some or all devices, respectively. For sample code that implements this request, please refer to the emberPermitJoiningRequest() API found in the "app/util/zigbee-framework/zigbee-device-common.h" file from your EmberZNet PRO installation.

Once the network contains at least one node within range of the joining device that permits joining, the joining device should be able to detect it as joinable through the stack's native emberNetworkFoundHandler() / ezspNetworkFoundHandler() callback or its emberJoinableNetworkFoundHandler() callback provided by the form-and-join utilities found in app/util/common/form-and-join.h, which are used by the Application Framework architecture. (See the Application Framework's "Network Find" plugin for a recommended implementation.)

4.3.1.3 Avoiding unintended consequences in the commissioning process

Once your joining device does find a joinable network and attempts to join it, the application or the installer must determine if it is the "correct" network, meaning the intended one rather than some other, arbitrary PAN that happened to be within range and permitting joining. The joining and subsequent authentication process, which involves the acquisition of the current NWK layer encryption key for the PAN, can fail in a variety of ways, even when joining the intended network. Therefore, permanently excluding networks where a join was attempted but failure in joining/authenticating has occurred is not necessarily the best practice. Similarly, depending on the security expectations of your joining device, it may be possible for it to successfully join a network that really isn't the correct one at all, so permanently settling into a network simply because the stack sends an EMBER_NETWORK_UP signal, indicating that the device was successfully joined and authenticated into the network, may not be sufficient either. The appropriate criteria for determining whether the attempted network is the correct one varies based on your design requirements, especially where security is concerned.

If you are designing a device for use in a Smart Energy network, a complex pre-authorization process is required before the device can successfully enter the network. See Chapter 6 for more information. Assuming that the requirement for pre-authorization has been met in the target network, joining the wrong network accidentally should be virtually impossible as the joining device won't accept the NWK key delivery if it arrives unencrypted or encrypted with a different APS link key.

However, even with the Smart Energy security model the joining node may still need to account for the fact that an unreliable link or other communication problem, especially if it involves the PAN's trust center, may cause the delivery of the NWK key from the trust center to fail, even in the correct network. Thus, if a joinable network is detected but the subsequent joining and authentication fails with EmberStatus of EMBER_NO_NETWORK_KEY_RECEIVED (meaning the NWK key didn't arrive successfully), EMBER_JOIN_FAILED (which could signify that the Association Response for the join wasn't received successfully), or EMBER_NO_BEACONS (meaning that the Association Request on the chosen network failed to get an answer), you may want to retry the joining process on that PAN again either immediately or later, in case the first attempt failed due to some temporary disruption. If the joining or authentication process continues to fail on the chosen PAN, consider attempting joining a different joinable network, provided one is available to your device, as the failures may be an indication that this is simply the wrong network.

In networks utilizing a Home Automation security model with a common, preconfigured APS link key used to pass a randomly generated NWK key, there is significant risk of

joining the wrong network by accident if multiple joinable networks happen to exist within range of the joining device, as the security settings among these networks are common to nearly all HA networks rather than being unique for each incoming node. Note that it is possible for HA networks to use a different preconfigured link key, but this key must somehow be communicated to the new node prior to its joining the network. Thus, you should take extra care in your application design to ensure that, once you enter a PAN successfully, it is really the intended one. This typically involves some kind of "join and verify" process for each available network that accepts your device, which means sending some sort of well-defined over-the-air message with an expected response to indicate joining of the correct network; this response may be another over-the-air message or may be some kind of detectable behavior by another part of the system. An example of each method is described in Table 4-2, beginning when a new node joins the candidate network:

Table 4-2. Join-and-Verify Methods

Triggering Event	Success	Exception
Example 1		
Node sends broadcast ZDO Match Descriptor Request for one or more target clusters that are important to the node's operation.	ZDO Match Descriptor Response received from one or more remote nodes.	No matches found; node goes to the next network candidate.
Caveats: As this verification process detects only devices that support the desired clusters, it can succeed when the joining node has entered a network with the same kind of security model and same kind of cluster capabilities as the intended network. For example, the garage door opener may join an arbitrary network with a garage door to control, but not the intended one. If this happens, the system must somehow detect the condition and instruct the new node to leave the current network and find a different one. This process only succeeds when at least one other node with the matching services has already been joined to the network. This introduces a commissioning requirement for the order in which devices must be joined, which must be communicated to the installer.		
Example 2		
Step 1: Node sends a unicast ZDO Match Descriptor Request to the network coordinator (node ID 0x0000), trying to match a device with support for ZCL Identify server cluster.	Node receives a ZDO Match Descriptor Response from the coordinator.	No response; node goes to the next network candidate.
Step 2: Node sends a ZCL Identify command to the coordinator on the matched endpoint.	The coordinator identifies itself to the system in some detectable way.	Identification not received within an expected timeframe; system acts on the node so that it leaves the network and goes to the next candidate.
Caveats In addition to the required instructions for the system, this method depends on having the coordinator accessible whenever a node is joined to the network. This is generally the case, as the coordinator is typically fulfilling the role of trust center to provide central authentication for each new node. This method also requires having some system-accessible stimulus available on the joining node to enable it to change networks.		

Triggering Event	Success	Exception
Example 3		
PAN's Trust Center receives the `TrustCenterJoinHandler` callback for the joining device.	PAN's Trust Center notifies the system through visual or audible indications on the Trust Center or some dedicated user interface such as a networked PC.	System does not get the expected indication; system acts on the node so that it leaves the network and goes to the next candidate.
Caveats This example does not require that the joining node's application send any extra messages, which greatly simplifies the commissioning design for the joining node's application. However, it does rely on certain intelligence and capabilities at the PAN's trust center device. Use of this method involves cooperation with the designer of the expected trust center (coordinator) node for your system.		

The choice of one of the above methods, or some variant thereof, will likely depend on the capabilities of the devices in your system, the importance of multi-vendor interoperability in your design, the expected latency of the commissioning process, and the sophistication of the installers who will be commissioning your devices.

ZigBee provides a commissioning cluster in the ZCL, which facilitates over-the-air installation of certain commissioning parameters into a device. However, as of this writing neither the HA nor SE profile requires implementation of this cluster on client or server side in its device types, nor has its use been tested as part of the ZigBee interoperability test events for these profiles. Use of the commissioning cluster is only feasible in networks where you can ensure that the joining node has server-side support for the commissioning cluster, and that at least one device in the system has client-side support to send commissioning commands. Furthermore, because the commissioning cluster relies on ZigBee messaging, which necessitates being in a network in the first place, you would need to design a scheme for having your device join a temporary commissioning network where a commissioning tool exists that can provide the necessary parameters. While Ember's Application Framework does provide an implementation of the ZCL's commissioning cluster, discussion of implementation for that cluster is beyond the scope of this document.

4.3.1.4 Leave mechanism

Many designers put careful thought into the network selection process and then neglect to provide a way for the system to uninstall the device from the current network and then install it into a new network. As the commissioning examples above show, enabling some way for the device to manually or automatically initiate an `emberLeaveNetwork()` action, and possibly find a new network after the leave completes, is often necessary to facilitate successful installation and reinstallation of ZigBee devices in their intended networks.

If this cannot be implemented in the hardware or software of the joining device itself, the ZDO's Leave Request mechanism, which is acted on automatically by the stack, may be a viable alternative, as it allows another node in the PAN, such as the network's controller, to instruct a device to leave the network. For sample code implementing the ZDO Leave Request command, refer to the `emberLeaveRequest()` API found in the "app/util/zigbee-framework/zigbee-device-common.h" file from your EmberZNet PRO installation.

4.4 Device Discovery and Provisioning

Once you've joined your device to the correct network, it needs some way to be paired up with other nodes in the PAN that provide related services (in other words, client-side devices are paired up to one or more server-side side devices). This process of pairing together related devices in the PAN for communication at the application level is referred to as "provisioning". By contrast, "commissioning" deals with associating devices together for communication at the networking stack level. As you architect your design, consider how you will discover which and how many devices in your PAN provide the services (clusters) of interest and by what means you will provision those related devices to one another. Note that the actual provisioning process generally concludes with one or more of the involved devices each registering the partner device(s) into its binding table, its address table, or some custom storage mechanism designed to remember the provisioned partner, so that messages can be sent to that destination. .

4.4.1 When to Discover and Provision

Often, application designers craft their application to perform some kind of device discovery, and attempt to provision soon after the commissioning process for a device is complete (that is, just after it comes online). However, because devices join one at a time, meaning one side of the provisioning is generally online before the other side, you will likely need to have a mechanism initiated by software state machine logic, external interrupt, or some over-the-air stimulus, to initiate provisioning later in the device's lifetime in the network.

Different provisioning methods are described in the next sections, with advantages and disadvantages for each.

Note: If you are designing an application for use in an official, public ZigBee application profile (such as Home Automation), Ember recommends that you review the latest published revision of the appropriate ZigBee application profile specification (as obtained from http://www.zigbee.org) for your target design to ensure that any profile-specific requirements or best practices for provisioning are being met by your design.

4.4.2 End Device Bind Method

The End Device Bind method is a ZigBee-defined process for one-to-one provisioning made possible by the ZDO (a set of features intrinsic to the stack and accessible through endpoint 0). The process, which is not specific to end devices and may be used with nodes of any type, involves each side of the provisioning process sending a ZDO End Device Bind Request to the PAN's coordinator. Each request contains a set of input (server-side) cluster IDs and a set of output (client-side) cluster IDs, as well an application profile ID and an endpoint number. The coordinator then acts as a "matchmaker" for the two devices: for each cluster that appears in one device's input cluster list as well as the other device's output cluster list, the coordinator sends a ZDO Bind Request to both devices, binding the endpoints of each node together on the matching cluster. This can potentially result in several bindings being created on each of the two devices. For sample code that you can use to create your own End Device Bind Requests, please refer to the following files:

- For EZSP host applications: `ezspEndDeviceBindRequest()` in app/util/zigbee-framework/zigbee-device-host.h.
- For SoC applications: `emberEndDeviceBindRequest()` in app/util/zigbee-framework/zigbee-device-library.h.

- For designs utilizing Ember's Application Framework (on any platform): `emberAfSendEndDeviceBind()` in app/framework/include/af.h.

Advantages:

- Doesn't require the devices involved to do any device discovery or service discovery themselves, because the coordinator acts as matchmaker. This is ideal for devices that need to maintain a low duty cycle, such as end devices, or devices with limited provisioning intelligence.
- Sets up bindings in each node's non-volatile memory as a way to permanently save the provisioning settings between the devices.
- The same method can be used after a successful End Device Bind process with same pair of devices to unbind them from each other (de-provision them).

Disadvantages:

- Process has a 60-second timeout between first and second requests.
- Usually requires manual stimulus by the user to generate the End Device Bind Request so that it's timed within one minute of the other node's request.
- If a node implements multiple endpoints, the application needs to decide whether multiple provisioning methods need to be made available to the user to bind each of the different endpoints.
- The Bind Manager support on the coordinator, which is the "matchmaking" feature critical to allowing End Device Bind functionality in the PAN, is an optional feature in ZigBee networks. However, the Bind Manager capability is required for coordinator nodes in a compliant ZigBee Home Automation network, so HA devices are able to rely on this feature being available.
- Because the Bind Manager puts bindings on both client and server nodes for each matched cluster as part of a successful matching process, many bindings can be created. The method generally requires a larger binding table if provisioning involves many clusters.
- Since performing End Device Bind method twice with same nodes removes the bindings, user error can result in accidental removal of provisioned bindings.

4.4.3 Identify and Group method

This method can be used for one-to-one provisioning or one-to-many provisioning, between a single source device (usually a client for the clusters being provisioned) and one or more target devices (usually servers for the clusters being provisioned). It involves putting each of the target nodes into Identify mode either through reception of an Identify command from some device or through some external stimulus. The source node then sends an "Add Group If Identify" command (a required client command in the ZCL's Identify cluster) as a broadcast, such that all nodes currently in identification mode add themselves to the specified group through the groups table maintained by the ZCL Groups server cluster. Once the target devices belong to a single group, the source device can send multicasts to the group either directly (`EmberOutgoingMessageType` of `EMBER_OUTGOING_DIRECT`) or by creating a Multicast binding for the target group and then sending outgoing commands through that binding (`EMBER_OUTGOING_VIA_BINDING`).

Advantages:

- Allows provisioning a single device to multiple targets simultaneously.

- Doesn't require any special Bind Manager support on a single node, like the End Device Bind method does.
- Requires as few as one over-the-air message (the `Add Group If Identifying` command) if all target devices support local method of being placed into Identify mode.
- Can be performed over a long or short time interval, as `Identify Time` used in the Identify command can be set very small or large.
- Can get devices into identification mode so that they are ready for provisioning in this manner over the air if needed, so works with devices without local stimulus (buttons or other user interface).
- If the source node can't be told manually to send `Add Group If Identifying` command, the source node can be put into identification mode and added to group so that multicast binding is created automatically (as part of groups table logic), for use in communicating with targets.

Disadvantages:

- Target devices must support Groups server and Identify server clusters.
- Target devices may require local stimulus such as a button press to get into identification mode, unless another device in the system can be told to send Identify command to specific devices of interest.

4.4.4 Push Button Method

This method involves pressing a button on one or more of the devices to cause it either to emit a message that other devices can recognize as a signal to engage in provisioning with this device as appropriate, or to enter a state where receiving a particular message within a particular time window causes it to engage in provisioning with the sender. For example, a light switch needing connection to one or more lights could use a button press to enter a state where, for the next 30 seconds, any "Add to switch" messages sent by lights cause the switch to register a binding entry for those lights. Similarly, a button press (or other stimulus) could be used to cause these "Add to switch" messages to be transmitted by each light.

Advantages:

- Lots of flexibility in implementation as far as what actions occur on button press and how long certain provisioning states last, which in turn impacts how long the provisioning process is allowed to take.
- Can be used for one-to-one or one-to-many provisioning.
- Doesn't require involvement from any third party devices (those not on either side of the provisioning process).
- Allows user/installer to explicitly control the provisioning process through manual interaction.
- Doesn't need any special cluster support.
- May be used in conjunction with other provisioning methods that involve manual interaction, such as Identify and Group method or End Device Bind method.

Disadvantages:

- One or both sides of the provisioning require local stimulus, such as a button, to engage in this process.

- May involve proprietary messaging protocol (for example, the proprietary "Add to switch" message discussed in the example above) or application-specific behavior to accomplish, reducing chances of interoperability among vendors.
- Provisioning between the wrong devices could occur if multiple provisioning processes are taking place simultaneously (such as if multiple installers are performing push-button provisioning on the same network at the same time).

4.4.5 Device Advertisement Method

This method is one in which a device of interest advertises its presence through broadcast or multicast transmission to a large audience and then allows the recipients to provision themselves to the advertising device as appropriate. The device receiving the advertisement could also choose to send a response to the advertiser that might cause the advertiser to provision itself to the responding device in the other direction. When provisioning devices for use with ZCL messaging, the advertising device could be client or the server. Usually it makes more sense to have the advertiser be the device that is generally on the receiving end of communication, which is typically the server as most ZCL transactions are initiated by the client side.

This is the provisioning method illustrated in Ember's Sensor/Sink sample application (see Ember document 120-3031-000, *Advanced Application Programming with the Stack and HAL APIs*, for details on this sample application). Each Sensor node requires a Sink, and since the design assumes that relatively few Sink nodes exist in the network relative to the number of Sensor nodes, the Sink devices advertise their identity periodically (once per minute by default) through a "Sink Advertise" multicast intended for the Sensors. Sensors that have not yet been provisioned to a specific Sink respond with a "Sensor Select Sink" unicast message in an attempt to provision themselves to the Sink in question. If the Sink acknowledges the selection with a "Sink Ready" message, the provisioning will be complete. Those Sensors that already have a valid Sink will ignore the advertisement.

Note that this method generally works best in cases where the potential audience for the advertisement (those nodes that need to provision themselves to the advertiser) is much larger than the number of advertising devices (like the ratio of Sinks to Sensors in the aforementioned example). More advertisers means more traffic, and it means that many devices can achieve provisioning from a single advertisement. Also, since one advertisement may potentially have many respondents, it is strongly recommended that the advertiser act as a network concentrator to facilitate routing of unicast responses to the advertiser. See Section 4.5.1, Many-to-One Routing, for more information about concentrator support.

Advantages:

- Scales efficiently to larger network scenarios. One advertisement can initiate provisioning of many recipients.
- Provisioning can occur without manual intervention. One side is advertising periodically, so the other side just needs to listen until it hears the advertisement.
- Provisioning can occur quickly and with minimal "back and forth" between each side. Advertisement recipients could potentially provision themselves to the advertiser as soon as they hear the advertisement.
- Can avoid the need for IEEE address (EUI64) discovery on each side, provided that the advertisement includes the advertiser's EUI64.

Disadvantages:

- No standard, agreed-upon advertisement message or a well-defined response message exists, so any implementation of this method is likely to be proprietary and therefore not 100% interoperable.
- Regular advertisements do create broadcast traffic periodically, and broadcast traffic in ZigBee networks is limited to approximately 1 broadcast per second, on average, so the designer should take care to avoid advertising too frequently when many concentrators exist in the network. Ember's Sink application uses a fixed advertisement period, but this advertisement interval can be made dynamic such that the intervals get longer as the network grows larger.
- This method typically requires Many-To-One Routing (MTOR) support on the advertising device, which increases flash requirements (as well as RAM requirements if source routing is used in conjunction with MTOR) and necessitates performing Many-To-One Route Requests (MTORRs) periodically. These MTORRs also add to the network's broadcast traffic burden discussed in the previous point.
- Since broadcast advertisements aren't reliably received by sleepy devices, a proprietary scheme such as the one depicted in Ember's Sensor/Sink example may be required to ensure that sleepy end devices can be made aware of the advertiser's presence.

4.4.6 Match Descriptor Request Method

In this method, a device looking to discover a particular partner for provisioning queries one or more nodes through the ZDO Match Descriptor transaction to find a suitable provisioning partner based on the other nodes' descriptor information (application profile, device identifier, cluster IDs, and client/server support) for each of their endpoints. In a typical scenario involving this method, a client device for a particular cluster (cluster X), configured on application profile Y, sends out a ZDO Match Descriptor Request as a broadcast to the network, with the descriptor information specifying an endpoint with server support for cluster X on profile Y. All nodes that receive this request process the message automatically using code built into every standard ZigBee Pro stack, by attempting to match the queried endpoint description with an endpoint descriptor on one of their own endpoints. If one or more endpoints on a queried device match the requested criteria, the queried device responds with a unicasted ZDO Match Descriptor Response containing the list of endpoints that match the request. The device that performed the query can then parse the responses and decide which (and how many) of the potential partners it should provision itself with.

Advantages:

- Provides for pairing based on a specific set of cluster support criteria.
- Doesn't require interaction of a third party to facilitate pairing (as compared with End Device Bind Method).
- Uses standard ZigBee ZDO frames for query and response, allowing for an interoperable solution without special parsing required of the query and responses. Stack handles ZDO queries automatically.
- Query can be broadcast or unicast.

Disadvantages:

- Generally relies on broadcasts to find all nodes, which is not 100% reliable and consumes network bandwidth.

- When sent as a broadcast, the ZDO Match Descriptor Request can only be sent to the Rx-On-When-Idle broadcast address, meaning that sleepy (Rx off when Idle) end devices are not discoverable with this method.
- The application requires some internal logic or a user interface to evaluate the query respondents and decide to which and to how many devices it should provision itself.
- Match Descriptor Responses may not contain the sender's EUI64, so provisioning based on long addresses (rather than dynamic, 16-bit node IDs) may require an additional unicast ZDO IEEE Address Request transaction to query the EUI64 of the partner node.

4.4.7 Simple Descriptor Request Method

The Simple Descriptor Request method is similar to the Match Descriptor Request method in that it uses standard ZDO queries to inquire about the target node's endpoint configuration (profiles, clusters, server/client support, and so on). However, where the Match Descriptor Request is sent as a broadcast or a unicast to try matching cluster support criteria on the recipient devices, the Simple Descriptor Request is only sent as unicast to a specific endpoint of a target node, and produces a full list of supported client and server clusters available on that endpoint of the target. This request can be used iteratively on each available endpoint of the target node to discover all possible cluster support available across endpoints. A ZDO Active Endpoints Request is frequently used as a precursor to the ZDO Simple Descriptor transactions so that the querying device can know how many valid endpoints exist at the target node and their endpoint numbers. Although this method yields more complete information than the Match Descriptor Request method, it is also less efficient, and so is generally not practical to use when a large number of devices need to be queried. This method is most useful when the querying device isn't sure which clusters it should provision with the target device; once it knows which clusters are available on the partner device, it can choose from among those clusters as it completes the provisioning process.

Advantages:

- Provides for pairing based on a specific set of cluster support criteria.
- Doesn't require interaction of a third party to facilitate pairing.
- Uses standard ZigBee ZDO frames for query and response, allowing for an interoperable solution without special parsing required of the query and responses. Stack handles ZDO queries automatically.
- Doesn't rely on broadcast mechanism.
- Doesn't require that the querying device knows which clusters are expected on the target device.

Disadvantages:

- Requires lots of back-and-forth transactions (one command/response transaction per endpoint, plus one for the Active Endpoints Request to assess viable endpoint numbers) to discover endpoint data of the target. While bandwidth consumption is relatively low for these transactions, the latency in completing the provisioning process can be significant, especially if there are multiple endpoints on the target.
- The application requires some internal logic or a user interface to decode which nodes to query and what to do with the response data when it arrives.

- Simple Descriptor Responses may not contain the sender's EUI64, so provisioning based on long addresses (rather than dynamic, 16-bit node IDs) may require an additional unicast ZDO IEEE Address Request transaction to query the EUI64 of the partner node. Note: Many devices will include their EUI64 in ZDO responses when the request is a unicast with the Source EUI64 APS option enabled, which is the default behavior for the EmberZNet PRO stack.

4.4.8 Provisioning Tool method

In this method, a third party device (not one of the nodes being provisioned) obtains information about some or all of the devices in the network and then provides a user interface to allow the network's installer/administrator to provision devices to one another in whatever manner he deems appropriate. This device information may be acquired through one of the ZDO discovery processes described in the preceding sections (Simple Descriptor Request or Match Descriptor Request) or by some more proprietary means (similar to the way device information is provided by a network concentrator in the Device Advertisement method).

The provisioning tool may be a device dedicated to this role, or may also be fulfilling other central roles in the network, such as network concentrator, PAN coordinator, commissioning tool, or gateway. Since the provisioning tool is likely to be communicating with a lot of different devices in the network, Ember recommends making this node behave as a network concentrator so that routes to the other nodes are readily available without requiring a series of route discoveries, which can burden the network and increase latency.

Once the tool decides which devices should be provisioned to one another, it typically uses a ZDO Bind Request to the target devices to install a binding table entry for communication to the partner device.

Advantages:

- Provides for pairing based on user input for maximum flexibility in provisioning.
- Doesn't require any intelligence beyond standard ZDO support in stack in provisioned devices for information discovery. All intelligence resides in provisioning tool node.
- Provisioned devices don't need to be awake to do their own provisioning, so the provisioning process can occur while one of the provisioned devices is asleep.
- Devices don't need to worry about re-discovering nodes for provisioning when new nodes enter the network, since the tool can take care of this.

Disadvantages:

- Requires a special tool (either a dedicated device or an extra set of functionality on an existing device) with a custom user interface to facilitate provisioning.
- The provisioning tool periodically needs to gather information from devices to avoid conflicting with the devices' own provisioning behaviors.

4.4.9 Address Discovery

Since provisioning often involves the creation of a binding table or address table entry, which relies on an EUI64 for tracking devices, the nodes involved in provisioning should make an effort to obtain the EUI64 of their partner devices to facilitate these table entry creations. For example, when using the Device Advertisement method, it is useful

to include the advertiser's EUI64 to prevent the receiver from having to discover this later, should the receiver choose to provision itself to the advertiser. As another example, when using ZDO request methods such as Simple Descriptor Request and Match Descriptor Request methods, the request should, when possible, contain the source EUI64 of the requesting device so that the receiver can respond in kind by providing its EUI64 address in the response frame, allowing for smoother provisioning.

4.5 Route establishment

4.5.1 Many-to-One Routing

In many networks, a large amount of data is funneled to a single node that is designated to store the data or offload it to another system or network. This behavior is most common in large sensor networks where information is gathered from many devices and aggregated at some central point.

Many-to-One Routing (MTOR) allows an aggregation point (a "concentrator" in ZigBee terminology) to provide every device on the network with a route to the concentrator without each node needing to discover it individually. Furthermore, MTOR provides a means to convey to the concentrator each node's own route to that concentrator. This allows the concentrator to collect some or all of these route records at its discretion, a technique known as "source routing". MTOR works in conjunction with source routing to allow bidirectional communication between the concentrator and other nodes without requiring discovery of new or updated routes at the time of message delivery.

4.5.1.1 Background - Many to One Routing

Early in ZigBee's[1] development, it became clear that a common communication pattern in embedded wireless networking applications was many-to-one, in which up to hundreds of devices might be communicating with a central gateway. At Ember, we sometimes use the term "aggregation" to refer to this pattern and the term "aggregator" for the gateway node.

4.5.1.2 How it Works in ZigBee Pro

This section briefly covers the details of how aggregation is now specified in the ZigBee Pro network layer.

The concentrator (for example, a gateway) establishes routes to itself by sending a many-to-one route request. This is just a regular route request sent to a special broadcast address. It signals the network layer of receiving nodes to create the inbound routes rather than a point-to-point route. No route replies are sent; the route record command frame described below serves a conceptually similar purpose.

When a device sends a unicast to the concentrator, the network layer transparently takes care of sending a route record command frame to the concentrator first. As the route record packet is routed to the concentrator, the relay nodes append their short IDs to the command frame. By storing the route obtained from the route record payload, the concentrator is supplied with the information it needs to source route packets in the reverse direction.

[1] See the ZigBee specification, document #053474. Sections of note include: 3.4.1.9 Source Route Subframe Field, 3.5.5 Route Record Command, 3.7.3.3.1 Originating a Source Routed Data Frame, and 3.7.3.3.2 Relaying a Source Routed Data Frame.

Source routing is accomplished by adding a subframe to the network frame, and setting a bit in the network frame control field. Upon receipt by relays, the next hop is read from the subframe rather than the local routing table. An application callback on the concentrator inserts the source route subframe into outgoing unicasts or APS acknowledgements as necessary.

Route maintenance is accomplished by the concentrator application resending the special many-to-one route request.

4.5.1.3 More Information

You can find additional information in the online API reference guide and in the many FAQ articles available on the Ember Developer's Portal, accessed through http://www.ember.com/support_index.html.

4.6 Message Delivery

This section provides an overview of this topic. If you would like more detailed information, please refer to Ember document 120-3031-000, Advanced Application Programming with the Stack and HAL APIs.

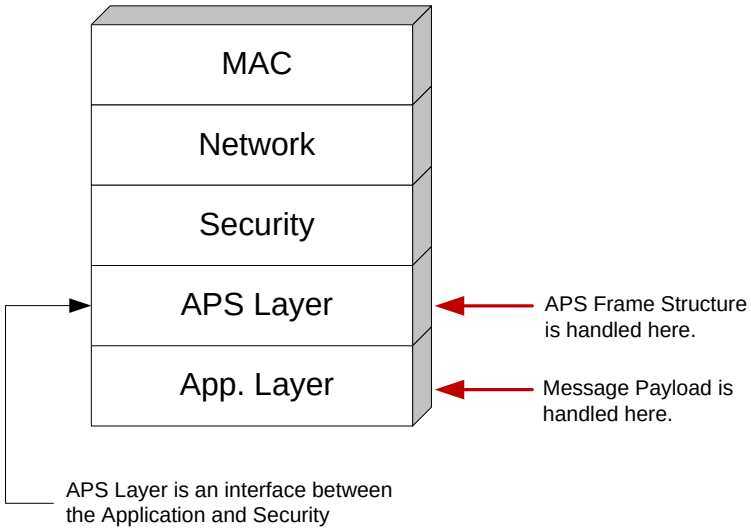
4.6.1 Message Handling

Message handling differs depending on whether you are using the SOC or NCP model and whether you are using the Application Framework or using the direct EmberZNet PRO API's. However, regardless of the model, many of the details and decisions involved in message handling are similar. Generally, message handling falls into two major tasks:

- Create a message
- Process incoming messages

The EmberZNet PRO stack software takes care of most of the low level work required in message handling. Figure 4-2 illustrates where the application interacts with the system in message handling. However, while the APS layer handles the APS frame structure, it is still the responsibility of the application to set up the APS header on outbound messages, and to parse the APS header on inbound messages.

Figure 4-2. Application/System Relationship During Message Handling



4.6.1.1 Sending a Message

Three basic types of messages can be sent:

- Unicast — sent to a specific node ID based on an address table entry (the node ID can also be supplied manually by the application if necessary)
- Broadcast — sent to all devices, all non-sleepy devices, or all non-ZEDs
- Multicast — sent to all devices sharing the same Group ID

Before sending a message you must construct a message. The message frame varies according to message type and security levels. Since much of the message frame is generated outside of the application, the key factor to be considered is the maximum size of the message payload originating in your application.

Table 4.2 shows the detailed API for sending the most common message types.

Table 4-3. API Messaging Functions

Ember ZNet API	Application Framework API	Description
`emberSendUnicast ( EmberOutgoingMessageType type, int16u indexOrDestination, EmberApsFrame * apsFrame, EmberMessageBuffer message )`	`emberAfSendUnicast( EmberOutgoingMessageType type, int16u indexOrDestination, EmberApsFrame *apsFrame, int8u messageLength, int8u* message )`	Sends a unicast message as per the ZigBee specification.

Ember ZNet API	Application Framework API	Description
`emberSendBroadcast ( EmberNodeId destination, EmberApsFrame * apsFrame, int8u radius, EmberMessageBuffer message )`	`emberAfSendBroadcast ( int16u destination, EmberApsFrame *apsFrame, int8u messageLength, int8u* message )`	Sends a broadcast message as per the ZigBee specification. The message will be delivered to all nodes within radius hops of the sender. A radius of zero is converted to EMBER_MAX_HOPS.
`emberSendMulticast ( EmberApsFrame * apsFrame, int8u radius, int8u nonmemberRadius, EmberMessageBuffer message )`	`emberAfSendMulticast( int16u multicastId, EmberApsFrame * apsFrame, int8u messageLength, int8u* message )`	Sends a multicast message to all endpoints that share a specific multicast ID and are within a specified number of hops of the sender.

Note: Please keep in mind that the online API documentation is more extensive than that shown here. Always refer to the online API documentation for definitive information.

In every case illustrated above, a message buffer contains the message. Normally, the application allocates memory for this buffer (as some multiple of 32 bytes). You can find out dynamically how big this buffer can be, which in turn determines the maximum size of the message to be sent. The function `emberMaximumApsPayloadLength(void)` returns the maximum size of the payload that the application support sub-layer will accept, depending on the security level in use. This means that:

1. Constructing your message involves supplying the arguments for the appropriate message type `emberSend...` function.
2. Use `emberMaximumApsPayloadLength(void)` to determine how big your message can be.
3. Executing the `emberSend...` function causes your message to be sent.

Normally, the `emberSend...` function returns a value. Check the online API documentation for further information.

While the task of sending a message is a bit complex, it is also very consistent. The challenge in designing your application is to keep track of the argument values and the messages to be sent. Some messages may have to be sent in partial segments and some may have to be resent if an error occurs. Your application must deal with the consequences of these possibilities.

4.6.1.2 Receiving Messages

Unlike sending messages, receiving messages is a more open-ended process. The application is notified when a message has been received, but the application must decide what to do with it and how to respond to it. Non-Application Framework-based applications use the stack's generic `emberIncomingMessageHandler()` to receive and handle messages. Application Framework-based applications use a variety of different callback functions devoted to the specific command or response represented by the message, such as `emberAfReadAttributesResponseCallback` or

`emberAfDemandResponseLoadControlClusterLoadControlEventCallback`. See Ember document 120-3028-000, *Application Framework V2 Developers Guide*, for details about how the Application Framework processes incoming messages and about the available callbacks for handling these in your application.

It is also important to note that the stack doesn't detect or filter duplicate packets in the APS layer. Nor does it guarantee in-order message delivery. These mechanisms need to be implemented by the application.

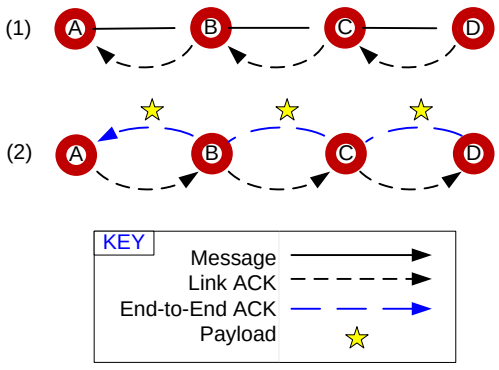
In the case of the SoC, the stack deals with the mechanics of receiving and storing a message. But in the case of the NCP, the message is passed directly to the host. The host must deal with receiving and storing the message in addition to reacting to the message contents.

In all cases, the application must parse the message into its constituent parts and decide what to do with the information. Note that the Application Framework, as part of the application, performs a majority of the message parsing. The Application Framework does still give the application developer complete flexibility and control over message receive processing. Messages can be generally divided into two broad categories: command or data messages. Command messages involve the operation of the target as a functional member of the network (including housekeeping commands). Data messages are informational to the application, although they may deal with the functionality of a device with which the node is interfaced, such as a temperature sensor.

4.6.1.3 Message Acknowledgement

When a message is received, it is good network protocol to acknowledge receipt of the message. This is done automatically in the stack software at the MAC layer with a Link ACK, requiring no action by the application. This is illustrated in Figure 4-3 (1) where node A sends a message to node D. However, if the sender requests an end-to-end acknowledgement, the application may want to add something as payload to the end-to-end ACK message (see Figure 4-3 (2)). Appending payload to APS ACKs is an Ember-proprietary feature and is not compatible with nodes running non-Ember stack software. See Ember document 120-3028-000, *Application Framework V2 Developer Guide*, for more information about adding payload to APS ACKs.

Figure 4-3. Link ACD and End-to-End ACK



Both Application Framework-based and non-Application Framework-based applications receive a callback when the delivery process completes (the callbacks have different names, as indicated in their respective reference manuals). The callback indicates success or failure of the delivery based on the receipt or lack of the ACK, either the APS [end-to-end] ACK if requested or the MAC [link] ACK if not.) Developers therefore can

read the success or failure of the delivery process and optionally retry the delivery att the application level, if desired.

4.7 Security

There are many security considerations when designing your application, such as the use of link keys, how keys will be derived, authentication policies, and so on. This topic has been given an entire chapter in this manual. For further information, see Chapter 6, Security Fundamentals.

5 Introducing the Ember HAL

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

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

5.1 HAL API Organization

The HAL API is organized into the following functional sections, which are described in section 5.4, HAL API Description:

- **Common microcontroller functions:** APIs for control of the MCU behavior and configuration.
- **Token access:** EEPROM, Simulated EEPROM (SimEEPROM), and Token abstraction. For a detailed discussion of the token system, see Chapter 8.
- **Peripheral access:** APIs for controlling and accessing system peripherals.
- **System timer control:** APIs for controlling and accessing the system timers.
- **Bootloading:** The use of bootloading is covered in Chapter 7.
- **HAL utilities:** General-purpose APIs that may rely on hardware capabilities (for example, CRC calculation that may take advantage of hardware acceleration).
- **Debug Channel**: API traces, debugging printfs, assert and crash information, and Virtual UART support when used with a DEBUG build of the EmberZNet PRO stack.

5.2 Naming Conventions

HAL function names have the following prefix conventions:

- `hal:` The API sample applications use. You can remove or change the implementations of these functions as needed.

- `halCommon:` The API used by the EmberZNet PRO stack and that can also be called from an application. Custom HAL modifications must maintain the functionality of these functions.
- `halStack:` Only the EmberZNet PRO stack uses this API. These functions should not be directly called from any application, as this may violate timing constraints or cause re-entrancy problems. Custom HAL modifications must maintain the functionality of these functions.
- `halInternal:` The API that is internal to the HAL. These functions are not called directly from the stack and should not be called directly from any application. They are called only from halStack or halCommon functions. You can modify these functions, but be careful to maintain the proper functionality of any dependent halStack or halCommon functions.

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

5.3 API Files and Directory Structure

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

- **<hal>/hal.h**: This master include file comprises all other relevant HAL include files, and you should include it in any source file that uses the HAL functionality. Most programs should not include lower-level includes, and instead should include this **top-level hal.h.**
- **<hal>/ember-configuration.c:** This file defines the storage for compile-time configurable stack variables and implements default implementations of functions. You can customize many of these functions by defining a preprocessor variable at compile-time and implementing a custom version of the function in the application. (For more information, see ember-configuration-defaults.h in the EmberZNet PRO API Reference for your platform.
- **<hal>/micro/generic**: This directory contains files used for general MCUs on POSIX-compliant systems. The default compiler is GCC.

EM250 HAL implementation

<hal>/micro/xap2b: This directory contains the implementation of the HAL for the XAP2b, which is the processor core used by the EM250. Functions in this directory are specific to the XAP2b but are not specific to the EM250 (see the next entry).

<hal>/micro/xap2b/em250: This directory implements functions that are specific to the EM250.

<hal>/micro/xap2b/em250/board: This directory contains header files that define the peripheral configuration and other PCB-level settings, such as initialization functions. These are used in the HAL implementations to provide the correct configurations for different PCBs.

EM35x HAL implementation

<hal>/micro/cortexm3: This directory contains the implementation of the HAL for the cortexm3, which is the processor core used by the EM35x. Functions in this directory are specific to the cortexm3 but are not specific to the EM35x (see the next entry).

<hal>/micro/cortexm3/em35x: This directory implements functions that are specific to the EM357.

<hal>/micro/cortexm3/em35x/board: This directory contains header files that define the peripheral configuration and other PCB-level settings, such as initialization functions. These are used in the HAL implementations to provide the correct configurations for different PCBs.

5.4 HAL API Description

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

5.4.1 Common microcontroller functions

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

5.4.2 Token access and Simulated EEPROM

EmberZNet PRO uses persistent storage to maintain manufacturing and network configuration information when power is lost or the device is rebooted. This data is stored in tokens. A token consists of two parts: a key used to map to the physical location, and data associated with that key. Using this key-based system hides the data's location from the application, which allows support for different storage mechanisms and the use of flash wear-leveling algorithms to reduce flash usage.

Note: For more information about the EmberZNet PRO token system, refer to both the token.h file and Chapter 8 of this manual.

Because the EM250, EM351, and EM357 do not contain internal EEPROM, a simulated EEPROM (also referred to as sim-eeprom and SimEE) has been implemented to use 8 kB of upper flash memory for stack and application token storage. Because the flash cells are only qualified for up to 1,000 write cycles in the EM2xx platforms (20,000 on the EM35x platforms), the simulated EEPROM implements a wear-leveling algorithm that effectively extends the number of write cycles for individual tokens into the tens to hundreds of thousands.

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

A status function is also available to provide basic statistics about the usage of the simulated EEPROM. For an in-depth discussion of the simulated EEPROM, its design, its usage, and other considerations, refer to Ember document 120-5037-000, *Using the Simulated EEPROM: For the EM250 SoC Platform*.

5.4.3 Peripheral access

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

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

5.4.4 System timer control

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

5.4.5 Bootloading

Bootloading functionality is also abstracted in the HAL interface. Refer to the EmberZNet PRO API reference for your platform as well as Chapter 7 of this manual for a detailed description on the use and implementation of the bootloaders.

5.4.6 HAL utilities

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

5.4.7 Debug channel

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

Note: Three levels of builds are provided: DEBUG provides API traces for EmberZNet PRO stack APIs along with other debug capabilities. NORMAL does not provide API traces, but provides everything else. The DEBUG_OFF variant has no SerialWire interfacing whatsoever, so has no Virtual UART and no ISD event tracing other than the Packet Trace Interface (PTI), which uses the ISA3 but doesn't rely on SerialWire.

On the EM250, the SIF interface on the InSight Port is used for the debug channel in addition to IDE level debugging. On the EM35x, the Serial Wire interface is used for debug channel in addition to IDE level debugging.

5.4.7.1 Virtual UART

EmberZNet PRO supports Virtual UART functionality with DEBUG and NORMAL builds. The Virtual UART allows normal serial APIs to still be used on the port being used by the debug channel for debug output. For the EM250, EM351, and EM357, which each only have a single physical UART numbered as port 1, the Virtual UART always occupies port 0. Virtual UART is automatically enabled when EMBER_SERIAL0_MODE is set to either EMBER_SERIAL_FIFO or EMBER_SERIAL_BUFFER.

When Virtual UART support is enabled, serial output sent to port 0 is encapsulated in the debug channel protocol and sent via the InSight Port. The raw serial output will be displayed by InSight Desktop, and will also appear on port 4900 of the adapter. Similarly, data sent to port 4900 of the adapter will be encapsulated in the debug channel protocol and sent to the node. The raw input data can then also be read using the normal serial APIs.

The Virtual UART provides an additional port for output with debug builds that would otherwise not be available.

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

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

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

5.4.7.2 Packet Trace support

EmberZNet PRO supports a PacketTrace interface for use with InSight Desktop. This capability allows InSight Desktop to see all packets that are received and transmitted by all nodes in a network with no intrusion on the operation of those nodes. The PacketTrace interface works with both the dev0455 and dev0680 Ember developer kit carrier boards running any application.

Custom nodes must have an InSight Port to use Packet Trace functionality. In addition to the proper hardware connections to use Packet Trace functionality, the BOARD_HEADER must define the PACKET_TRACE macro. You can use the settings in dev0455.h for EM250 or dev0680.h for EM351/EM357 as a template

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

5.5 Customizing the HAL

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

5.5.1 Compile-time configuration

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

Required definitions
The following preprocessor definitions must be defined:

- `PLATFORM_HEADER`: The location of the platform header file. For example, the EM357 uses hal/micro/cortexm3/em35x.
- `BOARD_HEADER:` The location of the board header file. For example, the EM357 developer board uses hal/micro/cortexm3/em35x/board/dev0580.h. Custom boards should change this value to the new file name.
- `PLATFORMNAME`, such as XAP2B or CORTEXM3.
- `PLATFORMNAME_MICRONAME` (for example, XAP2B_EM250 or CORTEXM3_EM357).
- `PHY_PHYNAME` (for example, PHY_EM250 or PHY_EM3XX.
- `BOARD_BOARDNAME` (for example, BOARD_DEV0455 or BOARD_DEV0680).
- `CONFIGURATION_HEADER`: Provides additional custom configuration options for ember-configuration.c.

Optional definitions
The following preprocessor definitions are optional:

- `APPLICATION_TOKEN_HEADER:` When using custom token definitions, this preprocessor constant is the location of the custom token definitions file.
- `DISABLE_WATCHDOG:` This preprocessor definition can completely disable the watchdog without editing code. Use this definition very sparingly and only in utility or test applications, because the watchdog is critical for robust applications.
- `EMBER_SERIALn_MODE = EMBER_SERIAL_FIFO` or `EMBER_SERIAL_BUFFER` (n is the appropriate UART port). Leave this undefined if this UART is not used by the serial driver. Note that the Buffer serial mode also enables DMA buffering functionality for the UART.
- `EMBER_SERIALn_TX_QUEUE_SIZE` = the size of the transmit queue in bytes (n is the appropriate UART port). This parameter must be defined if `EMBER_SERIALn_MODE` is defined for this UART port. In FIFO mode, the value of this definition specifies the queue size in bytes. In Buffer mode, the definition represents a queue size as a number of packet buffers, each of which is `PACKET_BUFFER_SIZE` bytes (32 bytes as of this writing).
- `EMBER_SERIALn_RX_QUEUE_SIZE` = power of 2 <= 128 (n is the appropriate UART port). Must be defined if `EMBER_SERIALn_MODE` is defined for this UART port. This value is always quantified in bytes (even in Buffer mode).
- `EMBER_SERIALn_BLOCKING` (n is the appropriate UART port). This must be defined if this serial port uses blocking IO (note that Ember does not recommend this for most applications).

5.5.2 Custom PCBs

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

You can use the EM357 developer kit carrier board header file dev0680.h as a template when creating a new board header. Modify the port names and pin numbers used for peripheral connections as appropriate for the custom board hardware. These definitions can usually be easily determined by referring to the board's schematic.

Once the new file is complete, change the preprocessor definition BOARD_HEADER for this project to refer to the new filename.

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

- `halInternalInitBoard`
- `halInternalPowerDownBoard`
- `halInternalPowerUpBoard`

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

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

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

In these cases, refer to the next section.

5.5.3 Modifying the default implementation

The functionality of the EmberZNet PRO HAL is grouped into source modules with similar functionality. These modules—the source files—can be easily replaced individually, allowing for custom implementations of their functionality. Table 5-1 summarizes the HAL source modules.

Table 5-1. EmberZNet PRO HAL Source Modules

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

Source Module	Description
uart	Low-level sample APIs used by the serial utility APIs to provide serial input and output

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

6 Security Fundamentals

6.1 Introduction

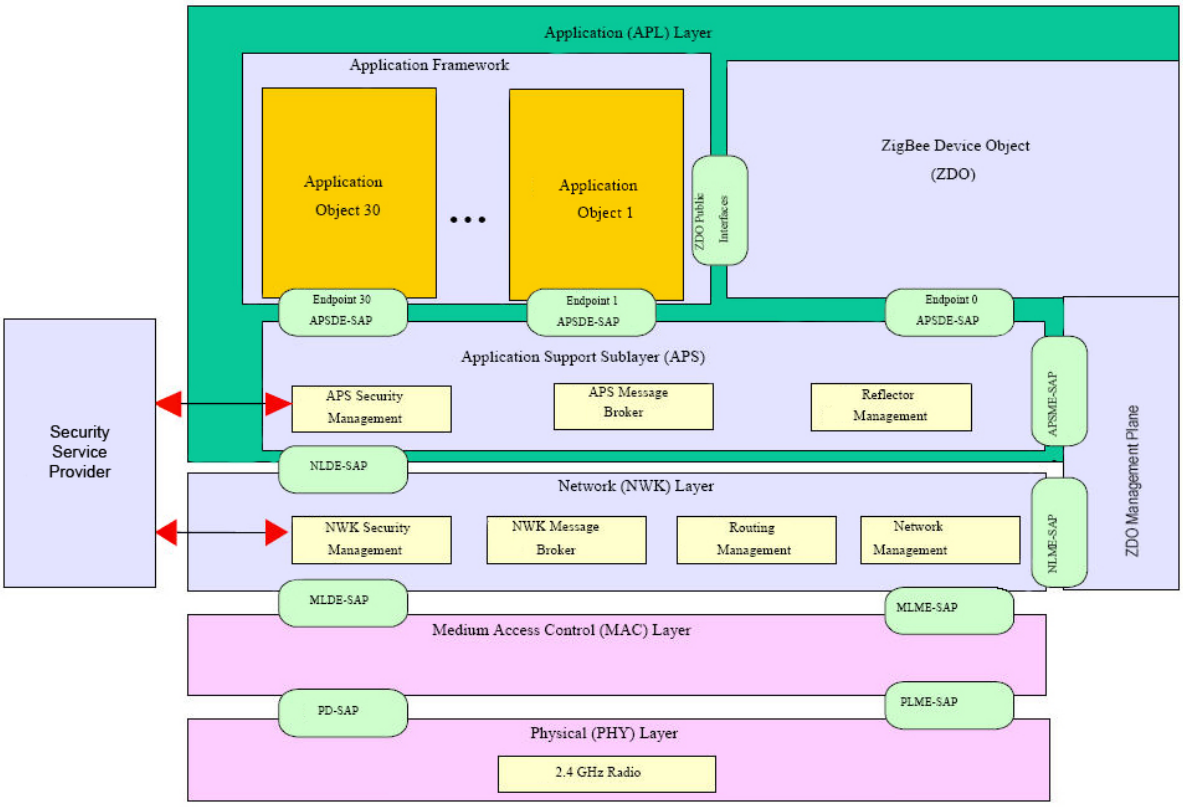
Security is a major concern in the ZigBee architecture. Although ZigBee uses the basic security elements in IEEE 802.15.4 (for example, AES encryption and CCM security modes), it expands upon this with:

- 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 6-1 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 6-1. ZigBee Stack Architecture



This chapter 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 Ember document 120-5070-000, *Smart Energy ECC-Enabled Device Setup*.

Those already familiar with ZigBee security concepts can jump to section 6.4, Implementing Security.

6.2 Concepts

6.2.1 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 6.2.3, APS Layer Security.

6.2.1.1 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.

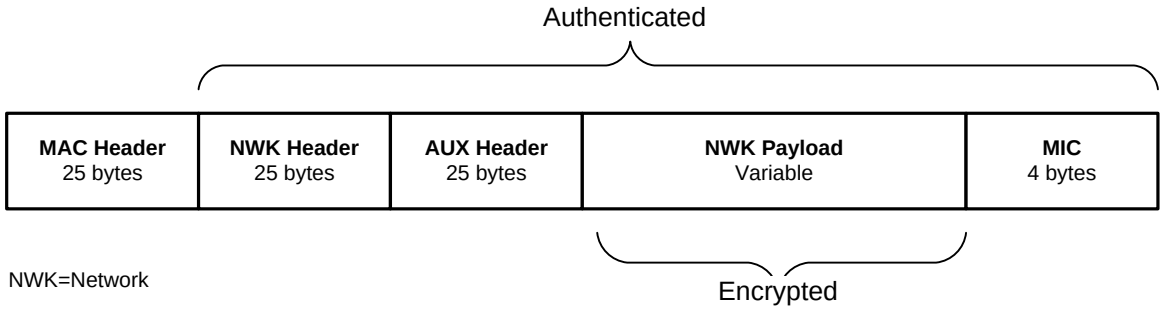
6.2.1.2 Hop-by-Hop Security

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.

6.2.1.3 Packet Security

A packet secured at the network layer is composed of the elements shown in Figure 6-2.

Figure 6-2. Anatomy of a Packet Secured at the Network Layer



6.2.1.4 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.

6.2.1.5 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.

6.2.1.6 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.

6.2.1.7 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.

6.2.2 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:

1. Send a copy of the current network key, which the parent relays to the joining device.
2. 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.
3. 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.

6.2.3 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.

6.2.3.1 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.

6.2.3.2 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.

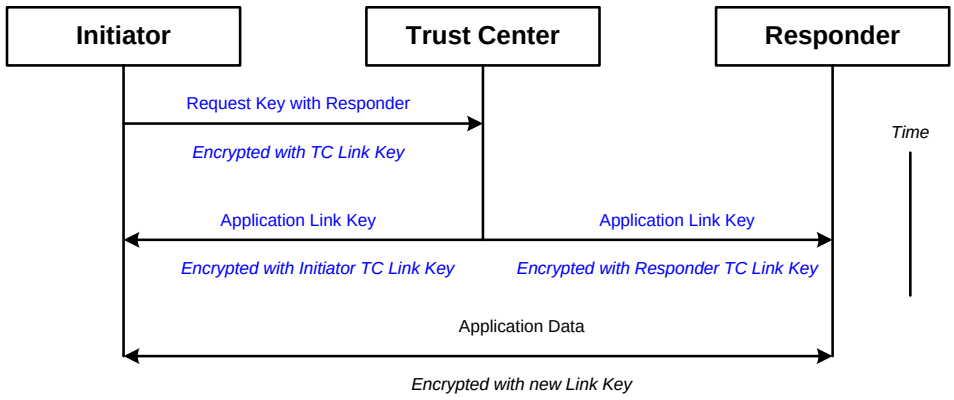
Standard security does not require trust center link keys, but devices may request one after joining. Ember highly recommends using trust center link keys. They are required for any device that wishes to rejoin a network of which it was previously a member.

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 6.3.2.3, Application Link Keys. The process for establishing an application link key is illustrated in Figure 6-3.

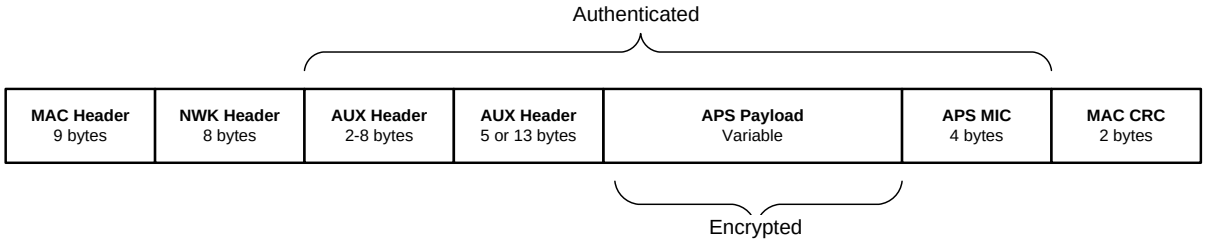
Figure 6-3. Establishing an Application Key



6.2.3.3 APS Packet Security

A packet secured at the APS Layer is composed of the elements shown in Figure 6-4.

Figure 6-4. APS Packet Security



6.2.3.4 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.

6.3 Standard Security

6.3.1 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 6.2.3, 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.

6.3.2 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.

6.3.2.1 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.

6.3.2.2 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).

6.3.2.3 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.

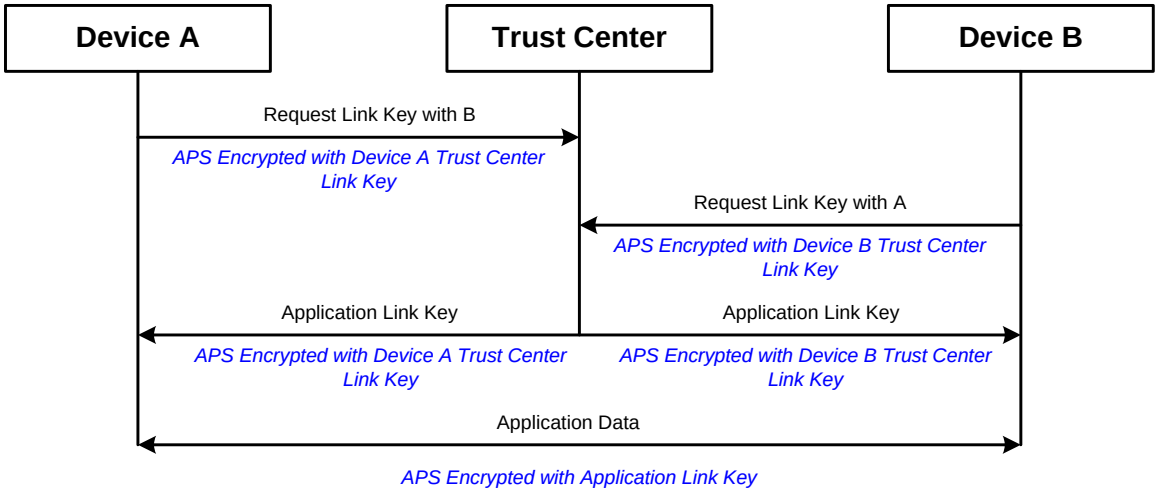
The application can manually configure a key by calling into the stack and setting one up. The partner device must also configure the application link key and negotiate with the other device when they can start using that key.

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 6-3, 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 6-5, 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 6-5. Requesting an Application Link Key with Another Device on the Network



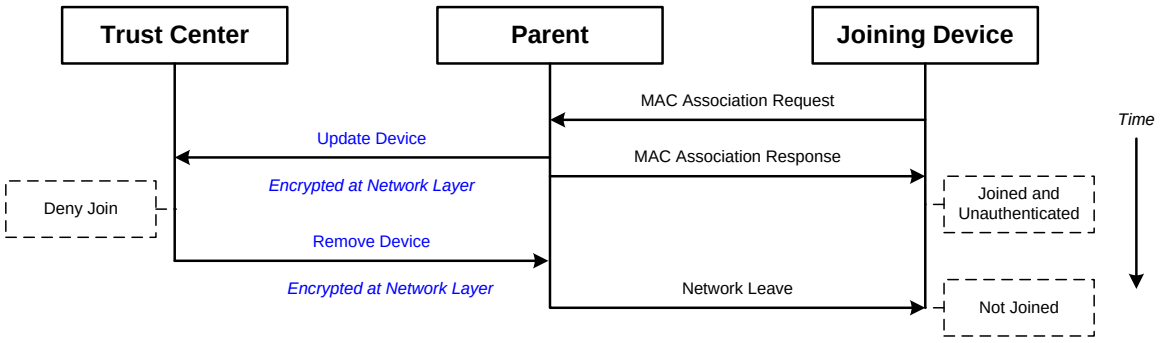
EmberZNet PRO supports a configurable table for storing application link keys. See section 6.4.4, Common Security Configurations, for more information.

6.3.3 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 6-6. If the device is allowed to join, the trust center's behavior depends upon whether the device has a preconfigured link key.

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



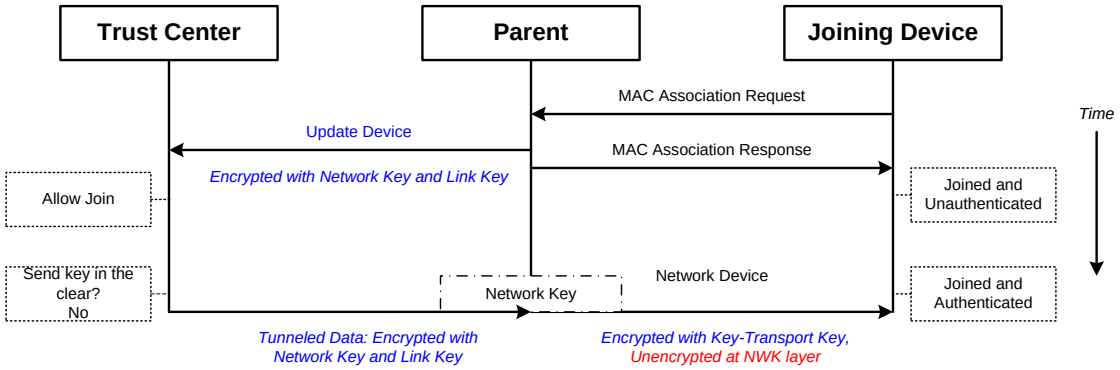
6.3.3.1 Preconfigured Link Keys

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 6-7 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 6-7. Joining Using a Preconfigured Trust Center Link Key



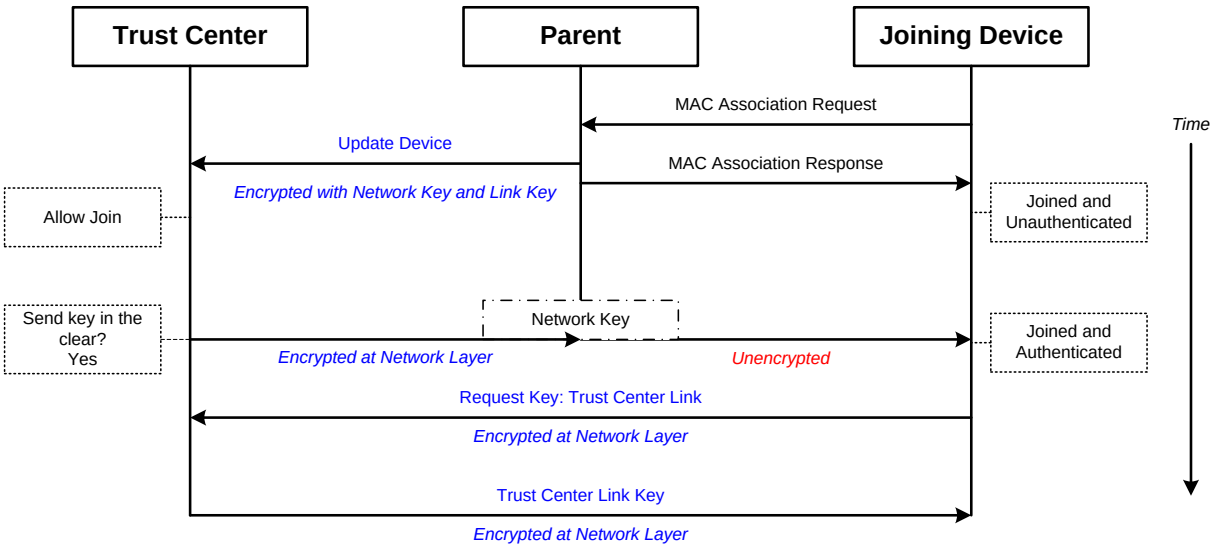
6.3.3.2 No Preconfigured Keys

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.

6.3.3.3 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 6-8.

Figure 6-8. Joining Without a Preconfigured Key and Requesting a 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.

6.3.4 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.

6.3.5 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.

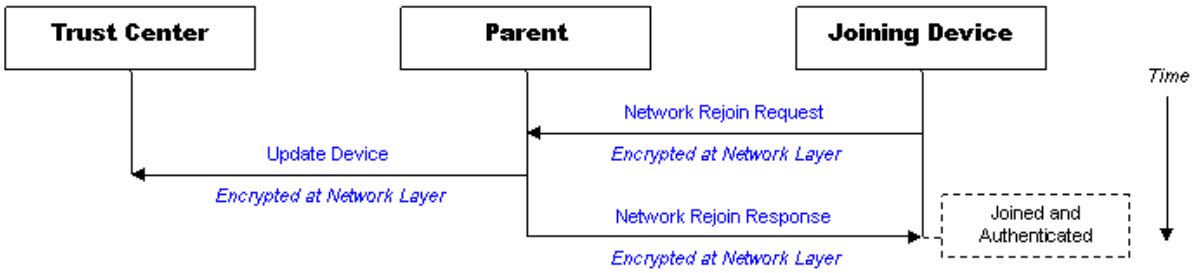
If the device has neither the current network key nor a trust center link key, it will have to perform a join.

6.3.5.1 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 6-9, 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 6-9. Secured Rejoin



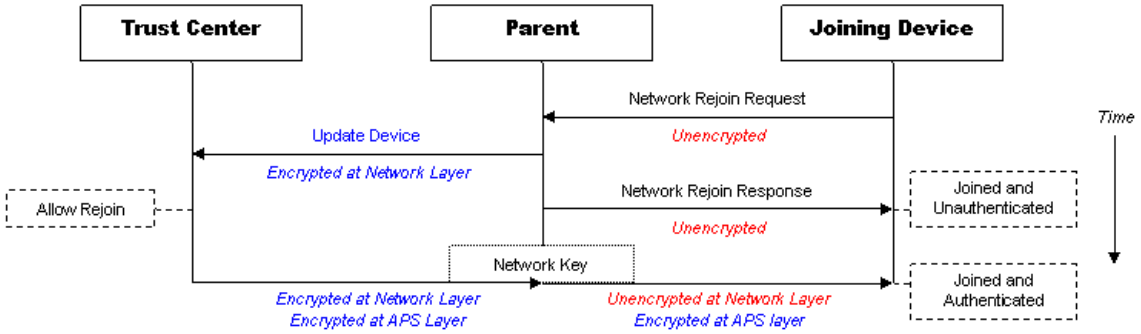
If the secured rejoin fails and the device is using standard security, the application can try an unsecured rejoin.

6.3.5.2 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 6-10, 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 6-10. Unsecured Rejoin

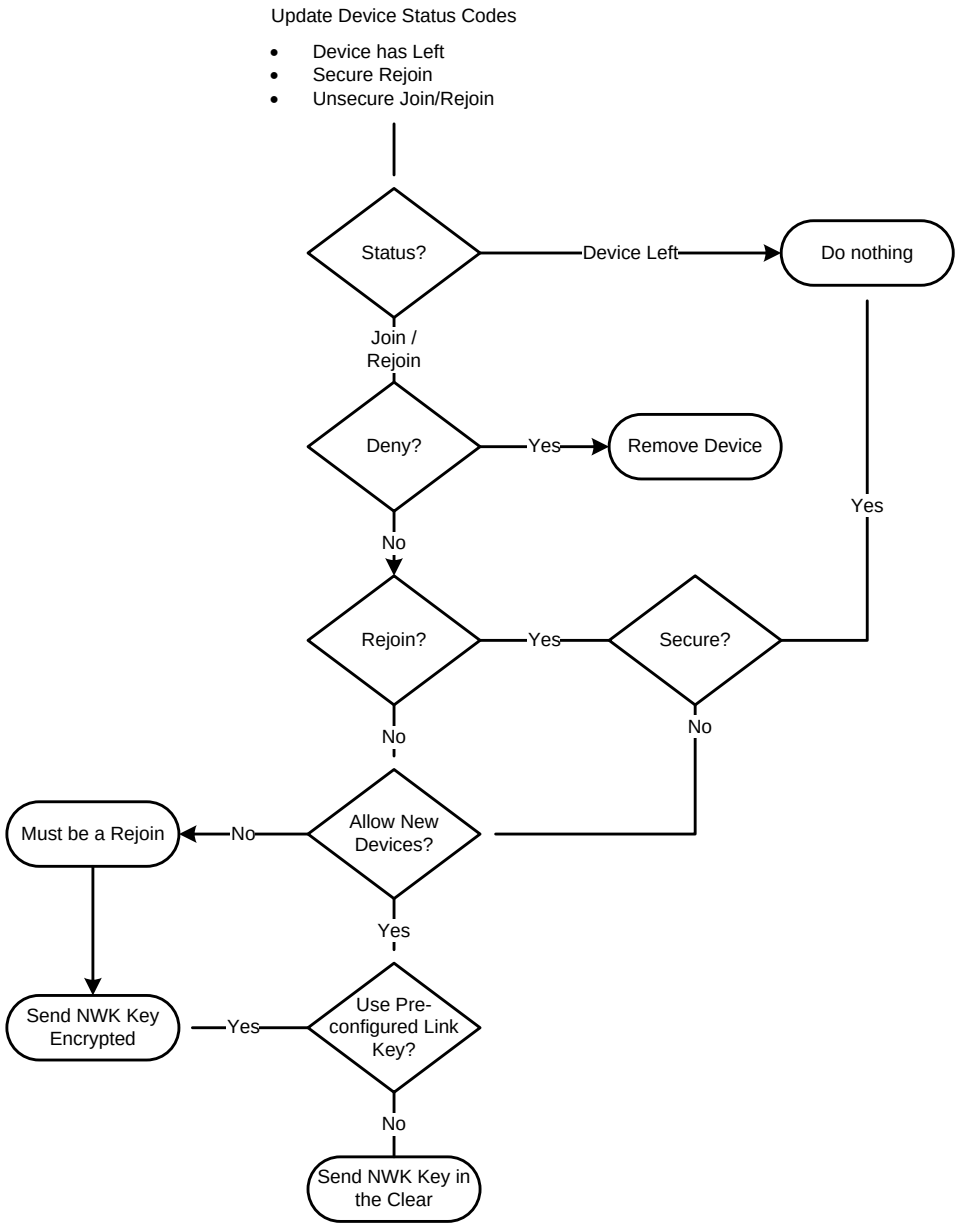


6.3.6 Trust Center Decision Process Summary

Figure 6-11 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 6-11. 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.

6.3.6.1 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.

6.3.6.2 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.

6.3.7 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.)

6.3.7.1 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.

6.3.7.2 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. Ember 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.)

6.3.7.3 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.

Ember provides sample code for implementing the trust center address cache mechanism on SoC platforms (See app/util/security/security-address-cache.c). Ember's 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 Ember document 120-3009-000, *EZSP Reference Guide,* for details on how to do this in your EZSP host implementation.)

6.4 Implementing Security

Security is implemented through stack software and message packet configuration. Security configuration in a typical application is often assisted by Ember's 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 6.4.4.10, Security Setup for ZigBee Home Automation (ZHA) Devices, and 6.4.4.11, Security Setup for ZigBee Smart Energy (ZSE) Devices, as well as section 6.5, 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 (Version 2) can be found in the Ember Document 120-3028-000, *AFV2 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.

6.4.1 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. Ember 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).

6.4.1.1 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.

6.4.2 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.

6.4.2.1 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 must be
set
   *  regardless of whether joining devices are expected to have a
preconfigured
   *  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;
```

6.4.2.2 Initial Security Bitmask

Table 6-1 describes the different settings used for security and the devices that may set them:

Table 6-1. 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.

Bits	Name	May be set by:	Description
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.
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().

Bits	Name	May be set by:	Description
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		

6.4.3 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.

6.4.3.1 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.

6.4.3.2 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.

6.4.3.3 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.

6.4.3.4 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.

6.4.3.5 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.

6.4.3.6 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 6.4.8, 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.

6.4.3.7 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.

6.4.3.8 Summary of Replay Protection

Table 6-2 provides a summary of the replay protection provided under the different security configurations supported.

Table 6-2. 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.

6.4.4 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.

6.4.4.1 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:

3. `emberNetworkInit(…)`
4. 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.

6.4.4.2 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.

6.4.4.3 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`.

6.4.4.4 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.

6.4.4.5 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 6.4.6, 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.

6.4.4.6 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.

6.4.4.7 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);
```

6.4.4.8 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);
```

6.4.4.9 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.

6.4.4.10 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 6.4.6, The Trust Center Join Handler, later in this chapter) 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.

6.4.4.11 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 6.5, ZigBee Smart Energy (ZSE) Security.

6.4.5 Error Codes Specific to Security

A number of error codes pertaining to security may be returned by the stack. Table 6-3 describes the error codes and when they may be received.

Table 6-3. 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.

Error Code	Description
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).

6.4.6 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:

1. The device left (EMBER_DEVICE_LEFT)
2. A secured rejoin was performed (EMBER_STANDARD_SECURITY_SECURED_REJOIN)
3. An unsecured join was performed (EMBER_STANDARD_SECURITY_UNSECURED_JOIN).
4. 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).

6.4.6.1 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 6-4.

Table 6-4. 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);
```

6.4.7 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)
```

6.4.7.1 Current Security State Structure

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

6.4.7.2 Current Security Bitmask

The bitmask of the current security settings is shown in Table 6-5.

Table 6-5. 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		

6.4.7.3 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)

6.4.7.4 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;
```

6.4.7.5 The Key Structure Bitmask

Table 6-6 describes the key structure bitmask.

Table 6-6. 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	-

6.4.8 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. The link keys library can be used in one of two ways:

1. 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.

6.4.8.1 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.

6.4.8.2 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.

6.4.8.3 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 6-7 contains the API calls pertaining to the link keys library.

Table 6-7. 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.

Link Key Table Function	Description
`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.

6.4.9 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).

5.4.9.1 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.

6.4.10 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.

6.4.10.1 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.

6.4.10.2 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(…)`.

6.4.11 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.

6.4.12 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 6.4.8, ZigBee Smart Energy (ZSE) Security.

The new trust center must also take into consideration the requirements presented in section 6.3.7, Additional Requirements for a Trust Center.

6.5 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.

6.5.1 Overview

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

6.5.1.1 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 6.4.4.11, 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.

6.5.1.2 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 6-8.

Table 6-8. 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 6-12. 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 6-12. 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 section6.3.2.3, Application Link Keys.

6.5.1.3 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.

Ember's Application Framework, Version 2, also known as "AFV2", can ensure that the above requirements are satisfied (where applicable) when Ember's AppBuilder tool is used to configure a ZSE device. For more information about AFV2-based development, refer to the Ember document 120-3028-000, *Application Framework V2 Developer Guide*.

6.5.2 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.)
- Ember document 120-5058-000, *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.ember.com website). It explains how to program the necessary manufacturing data (certificates and installation codes) into Ember chips using Ember's programming tools and how to verify these data once they have been programmed into the device.

- Ember document 120-5070-000, *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.
- Ember document 120-3028-000, *Application Framework V2 Developer Guide*. This document, included in EmberZNet PRO software releases and available online through the www.ember.com website, describes the command line interface (CLI) commands available for inspecting and altering the security configuration of an AFV2-based application at runtime. It also explains any callbacks or plugins used by AFV2 to implement security behavior in the application.
- Ember document #120-4035-000, *Ember AppBuilder User's Guide*. This document, included in the InSight 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 AFV2-based application.

7 Bootloading

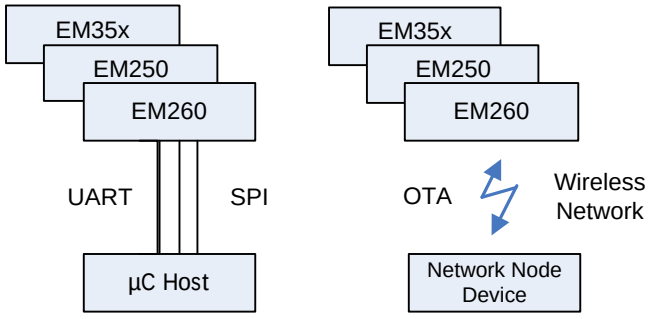
This chapter serves as an introduction to bootloading for Ember ZigBee networking devices. It looks at the concepts of standalone and application bootloading and discusses their relative strengths and weaknesses. In addition, it looks at design and implementation details for each method.

7.1 Introduction

The bootloader is a program stored in reserved flash memory that allows a node to update its image on demand, either by serial communication or over the air. Production-level programming is typically done during the product manufacturing process yet it is desirable to be able to reprogram the system after production is complete. More importantly, it is valuable to be able to update the device's firmware with new features and bug fixes after deployment. The bootloading capability makes that possible.

Bootloading can be accomplished through a hardwired link to the device or over the air (that is through the wireless network) as shown in Figure 7-1.

Figure 7-1. Bootloading Links



Ember's ZigBee networking devices offer two types of bootloaders: standalone and application. These two bootloaders can differ in the amount of flash required and location of the stored image, as discussed in the next two sections.

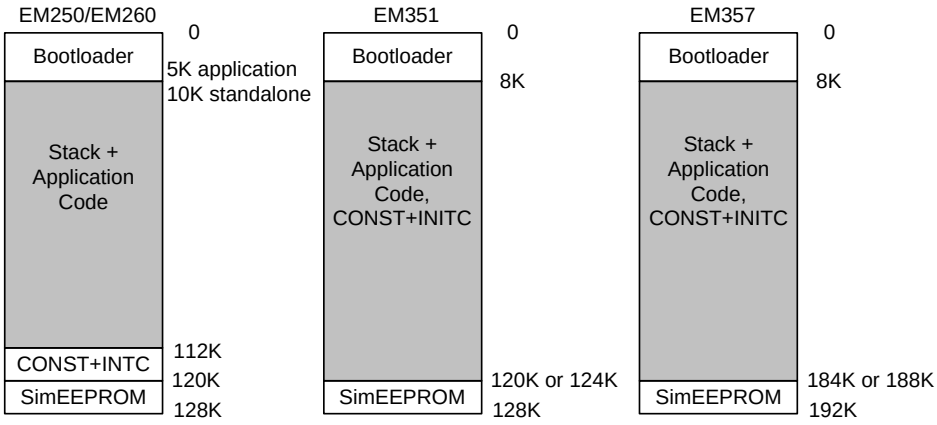
Ember supports devices that do not use a bootloader, but this requires external hardware, such as the InSight Adapter or USBLink, to upgrade the firmware. Devices without a bootloader have no supported way of upgrading the firmware over the air once they are deployed, which is why Ember strongly advocates implementing a bootloader.

The bootloading situations described in this document assume that the source node (the device sending the firmware image to the target through a serial or OTA link) acquires the new firmware through some other means. For example, if a device on the local ZigBee network has an Ethernet gateway attached, this device could get or receive these firmware updates over the Internet. This necessary part of the bootloading process is system-dependent and beyond the scope of this document.

7.1.1 Memory Space for Bootloading

Figure 7-2 shows the memory maps for each of Ember's ZigBee networking devices.

Figure 7-2. Ember ZigBee Networking Devices' Memory Maps

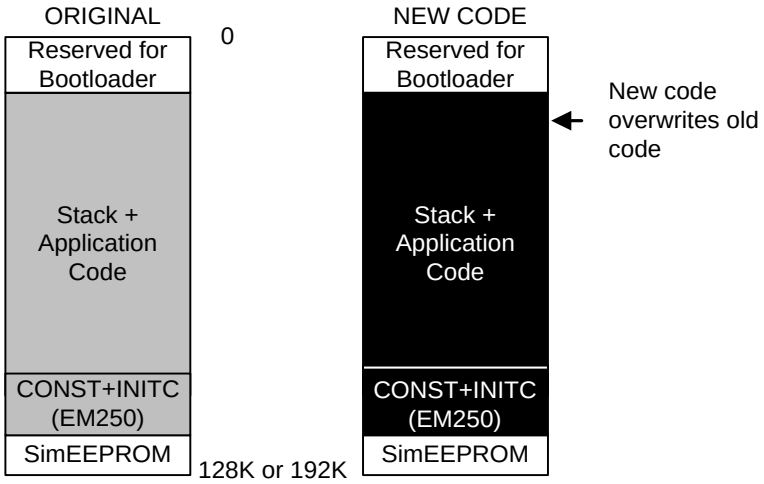


For the EM250 and EM260, a block of 10 kB of low flash memory is reserved to hold the bootloader and a block of 16 kB of high memory is reserved for a simulated EEPROM (8 kB) and CONST+INITC (8 kB). For the EM35x, a block of 8 kB of low flash memory is reserved to hold the bootloader, as well as either 4 kB or 8 kB of high flash memory for the simulated EEPROM. In all cases, the balance of the memory space is unreserved and available to hold EmberZNet PRO and application code.

7.1.2 Standalone Bootloading

Standalone bootloading is a single-stage process that allows the application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. Very little interaction occurs between the standalone bootloader and the application running in flash. In general, the only time that the application interacts with the bootloader is when it wants to run the bootloader, when it calls `halLaunchStandaloneBootloader()`. Once the bootloader is running, it receives bootload packets containing the (new) firmware image either by physical connections such as UART or SPI, or by the radio (over-the-air). Figure 7-3 illustrates what happens to the device's flash memory during bootloading.

Figure 7-3. Standalone Bootloading Code Space (Typical)

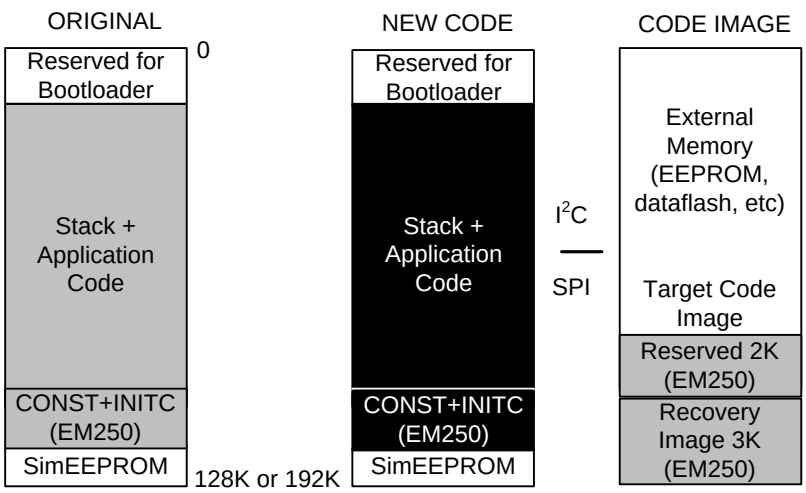


Note: When bootloading is initiated, the new code overwrites the existing stack and application code. If any errors occur during this process, the code cannot be recovered and bootloading must start over at byte zero.

7.1.3 Application Bootloading

The application bootloader relies on the application to perform the upgrade process. Application bootloading has three components: the application, the application bootloader, and the recovery image. An area is also reserved for storing the new target code image. This is often referred to as the download space, and usually requires an external memory device such as an EEPROM or dataflash. The new firmware is first loaded into the download space. This is typically handled by the application either using a UART serial connection or over-the-air. Once the new image has been stored, the application bootloader is then called to validate the new image and copy it from the download space to flash. The application bootloader does not participate in acquiring the image. Because the application bootloader does not need to operate the radio, it is much smaller than the standalone bootloader. Figure 7-4 shows a typical memory map for the application bootloader.

Figure 7-4. Application Bootloading Code Space (Typical)



Download errors do not adversely impact the current application image while storing the new image to the download space. The download process can be restarted or paused to acquire the image over time.

7.2 Design Decisions

Table 7-1 lists some major considerations for each type of bootloading. These are the trade-offs that must be considered by the systems designer.

Table 7-1. Design Trade-Offs

Standalone Bootloading	Application Bootloading
Self-contained	Additional hardware and application code required
Serial Link	Serial Link
OTA Link (restricted to single hop communication)	OTA Link (multi-hop capable)
Larger memory requirement for bootloader code (EM250 SoC platform only)	Smaller memory requirement for bootloader code (EM250 SoC platform only)
Must be run on its own to completion with the customer application disabled	Is a part of the customer application so customer code can still run while downloading a new image
Bootload errors generally render the application nonfunctional and require retransmission	Bootload errors can typically be repaired from the saved code image

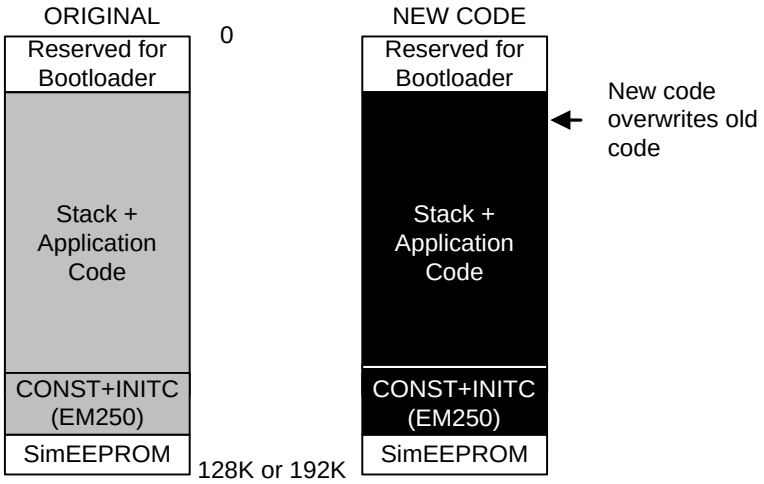
The next two sections provide detailed information about both types of bootloading.

7.3 Standalone Bootloading

7.3.1 Introduction

A standalone bootloader is a program that uses either the serial port or the radio in a single-hop, MAC-only mode to get an application image. Standalone bootloading allows the new application image to be placed into flash memory, overwriting the existing application image, without the participation of the application itself. Figure 7-5 shows a sample memory map and how the bootloading process overwrites the old code image. It should be clear from this illustration that the bootloading process is destructive and must proceed to completion if a functional application is to reside in this code space. A failure during bootloading means that the process must begin again. Bootloading for the EM35x and EM250 work similarly and are discussed together. The operating model of the EM260 is a little more complicated, but the basic idea behind bootloading an EM260 application is the same. Contact Ember support if you have specific questions about bootloading an EM260.

Figure 7-5. Standalone Bootloading Code Space (Typical)



Note: When bootloading is initiated, the new code overwrites the existing stack and application code. If any errors occur during this process, the code cannot be recovered and bootloading must start over at byte zero.

7.3.2 Modes - Serial / OTA

The standalone bootloader and its utility library support three basic modes for uploading an application image to a network device:

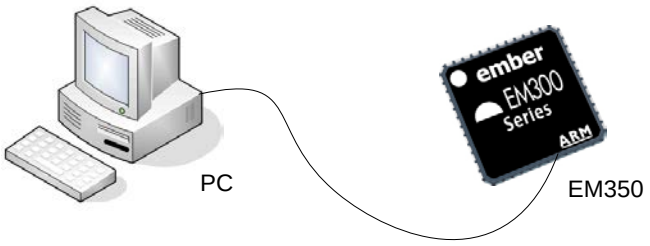
- Serial upload
- Over-the-air upload
- Hybrid mode uploads

7.3.3 Serial Upload

You can establish a serial connection between a PC and a target device's serial interface and upload a new software image to it using the XModem protocol, as shown in Figure 7-6. If you need information on the XModem protocol there is plenty of documentation

online. A good place to start would be http://en.wikipedia.org/wiki/XMODEM, which should have a brief description and up to date links to protocol documentation.

Figure 7-6. Serial Upload



A serial connection is established as follows:

1. The PC connects to the target device at 115,200 baud, 8 data bits, no parity bit, and 1 stop bit (8-N-1).
2. The target device's bootloader sends output over its serial port after it receives a carriage return from the PC. This prevents the bootloader from prematurely sending commands that might be misinterpreted by other devices that are connected to the serial port.
3. After the bootloader receives a carriage return from the target device, it displays a menu with the following options:

```
EM250 Bootloader v20 b01
1. upload ebl
2. run
3. ebl info
BL >
```

After listing the menu options, the bootloader's "BL >" prompt displays.

Note: Scripts that interact with the bootloader should use only the "BL >" prompt to determine when the bootloader is ready for input. While current menu options should remain functionally unchanged, the menu title and options text is liable to change, and new options might be added.

7.3.3.1 Serial Upload: Uploading an Image

Selection of the menu option 1 (upload ebl) initiates upload of a new software image to the target device, which unfolds as follows:

1. The target device awaits an XModem CRC upload of an EBL file over the serial line, as indicated by the stream of C characters that its bootloader transmits.
2. If no transaction is initiated within 60 seconds, the bootloader times out and returns to the menu.
3. After an image successfully uploads, the XModem transaction completes and the bootloader displays 'Serial upload complete' before redisplaying the menu.

7.3.3.2 Serial Upload: Errors

If an error occurs during the upload, the bootloader will display the message 'Serial upload aborted' followed by a more detailed message and one of the hex error codes shown in Table 7-2. It will then redisplay the bootloader menu.

Table 7-2. Serial Uploading Error Messages EM250 & EM35x

Hex code	Constant	Description
0x21	BLOCKERR_SOH	The bootloader encountered an error while trying to parse the start of header (SOH) character in the XModem frame.
0x22	BLOCKERR_CHK	The bootloader detected an invalid checksum in the XModem frame.
0x23	BLOCKERR_CRCH	The bootloader encountered an error while trying to parse the high byte of the CRC in the XModem frame.
0x24	BLOCKERR_CRCL	The bootloader encountered an error while trying to parse the low byte of the CRC in the XModem frame.
0x25	BLOCKERR_SEQUENCE	The bootloader encountered an error in the sequence number of the current XModem frame.
0x26	BLOCKERR_PARTIAL	The frame that the bootloader was trying to parse was deemed incomplete (some bytes missing or lost).
0x27	GOT_DUP_OF_PREVIOUS	The bootloader encountered a duplicate of the previous XModem frame.
0x41	BL_ERR_HEADER_EXP	No .EBL header was received when expected.
0x42	BL_ERR_HEADER_WRITE_CRC	Header failed CRC.
0x43	BL_ERR_CRC	File failed CRC.
0x44	BL_ERR_UNKNOWN_TAG	Unknown tag detected in .EBL image.
0x45	BL_ERR_SIG	Invalid .EBL header signature.
0x46	BL_ERR_ODD_LEN	Trying to flash odd number of bytes.
0x47	BL_ERR_BLOCK_INDEX	Indexed past end of block buffer.
0x48	BL_ERR_OVWR_BL	Attempt to overwrite bootloader flash.
0x49	BL_ERR_OVWR_SIMEE	Attempt to overwrite SIMEE flash.

Hex code	Constant	Description
0x4A	BL_ERR_ERASE_FAIL	Flash erase failed.
0x4B	BL_ERR_WRITE_FAIL	Flash write failed.
0x4C	BL_ERR_CRC_LEN	End tag CRC wrong length.
0x4D	BL_ERR_NO_QUERY	Received data before query request/response.
0x4E	BL_ERR_BAD_LEN	An invalid length was detected in the .EBL image.
0x4F	BL_ERR_TAGBUF	An invalid tag was found in the .EBL image.

7.3.3.3 Running the Application Image

Bootloader menu option 2 (run) resets the target device into the uploaded application image. If no application image is present, or an error occurred during a previous upload, the bootloader returns to the menu.

7.3.3.4 Obtaining Image Information

On the EM250, bootloader menu option 3 (ebl info) displays platform information about the uploaded image, in the following format:

platform-micro-phy-board

For example, the bootloader shows the following data about an image that is built for the Ember EM250 RCM and breakout board:

xap2b-em250-em250-dev0455

On the EM35x platform, the image info is customizable by the user, and can be specified as a string using the --imageinfo option in the em3xx_convert utility, which creates the EBL image from an s37 file. Menu option 3 then displays the information as a quoted string, similar to the following:

"custom image info"

For both the EM35x and EM250 the information displayed by these commands represents the image that is currently stored in the flash. This means that after a successful bootload this information should change to reflect the new application.

7.3.4 Over-the-Air Upload

You can upload images over the air to a target device in several ways:

- Passthrough
- Multi-hop Passthrough

In all cases, the source device must be within radio range of the target device. The uploaded image can originate from a PC or some other device that sends the image to a network device over a serial line.

The source device uses a simplified MAC-based protocol to communicate with the target, which can only travel one hop. This protocol is based on XModem CRC but uses 64-byte data blocks that can fit in a single 802.15.4 packet.

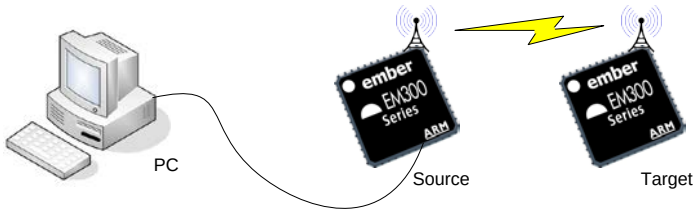
During over-the-air upload, only the target device actually runs the bootloader. The source device and any intermediary devices that participate in the upload process continue to run an application that is based on the EmberZNet PRO stack.

Note: If a target device gets a carriage return from its serial port while it awaits over-the-air bootloader packets from another device, and if no over-the-air bootloader-formatted packets have arrived, the device's bootloader switches to serial mode and ignores any subsequent over-the-air packets.

7.3.4.1 Passthrough

When using over-the-air passthrough mode, the source node is connected to a PC by a serial cable. The source receives an image over the serial line and passes the image to the target node over the air, as shown in Figure 7-7.

Figure 7-7. Over-the-air Passthrough Mode



The PC is expected to transfer the .EBL image to the source node using standard 128-byte XModem CRC packets. The source node application must split these packets into the special 64-byte XModem format that the Ember standalone bootloader uses for its over-the-air protocol.

7.3.4.2 Passthrough: Upload Process

As soon as the source node verifies that the target node is running the bootloader, it starts the upload process by calling `bootloadUtilStartBootload()` with the mode parameter set to `BOOTLOAD_MODE_PASSTHRU`.

After receiving the query response from the target node, the source node calls `emberIncomingBootloadMessageHandler()`, which directs the serial download by calling `XModemReceiveAndForward()`.

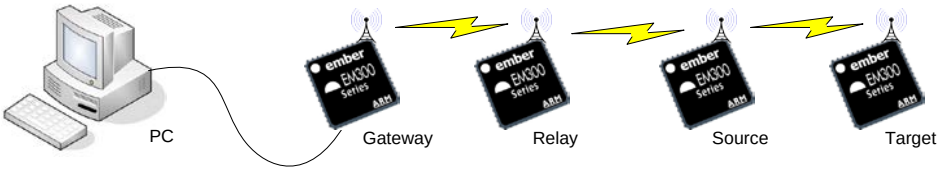
For more detailed information, see the sample code in the bootloader utility library (/app/util/bootload).

7.3.4.3 Multi-Hop Passthrough

Multi-hop passthrough mode is an extension of standard over-the-air passthrough, and is used when an image must travel longer distances, as shown in Figure 7-8. In this mode, a gateway device receives an image over the serial line, then uses standard networking protocols to forward the image to a network device. The device converts the ZigBee-

formatted messages to the bootloader's over-the-air link-layer protocol, then forwards the image to the target.

Figure 7-8. Over-the-air Multi-hop Passthrough Mode



7.3.5 Hybrid Mode Uploads

Upload operations can mix any of the modes previously described. For example, a source node might acquire an image by some other means—for example, from a set of images stored on an externally connected serial flash.

7.3.6 Upload Recovery

If an image upload fails, the target node is left without a valid application image. Typically, failures are related to over-the-air transmission errors. When an error occurs, the bootloader restarts and continues to listen on the same channel for any retries by the source node. It remains in recovery mode until it successfully uploads the application image.

If a hard reset occurs before the bootloader receives a new valid image, or the bootloader is launched by the hardware trigger, the target node enters bootload recovery mode. In this mode, the bootloader listens on the default channel (13) for a new upload to begin. The primary use of GPIO5 on the EM250 is PTI_DATA (Packet Trace Data) supplied to the InSight Adapter while the application is running. During module power up, holding GPIO5 low triggers the emergency recovery mode. Optionally, GPIO5 can be used as the ADC1 input. The EM250 Datasheet provides full details on the hardware functionality of GPIO5.

The EM260 uses the nWAKE and PTI_DATA pins to trigger recovery mode on power up. See Ember document 120-0260-000, *EM260 Datasheet* (chapter 5.6.1), for details.

The EM35x uses PA5 as a hardware-based trigger for recovery mode. As in the EM250, the primary use of this pin is PTI_DATA. Holding this pin low during power-up or across a reset and then sending a carriage return at 115200 baud launches the standalone bootloader. The EM35x platform can also be configured to use other IO pins or other schemes of activation by modifying the `bootloadForceActivation()` API in bootloader-gpio.c and rebuilding the bootloader from the provided project files. An example of utilizing PC6, which is connected to a button on the EM35x Breakout Board, is provided in bootloader-gpio.c and can be enabled by building the bootloader with `USE_BUTTON_RECOVERY` defined.

After the source node identifies a node that is in recovery mode, it resumes the upload process as follows:

1. The source application starts the download process by calling `bootloadUtilStartBootload()`. Before calling the function, the source node needs to ensure that it is on the same channel as the node to be recovered. For EM250 applications, the source node can leave the current channel and join or form the network on the recovering channel. For EM260 host applications, the source node can call `ezspOverrideCurrentChannel()` to change the channel to the recovering channel. In case of default channel recovery, the source node needs to be on bootload default channel (13).
2. The source node sends an XMODEM_QUERY message to the target node.
3. The target node bootloader extracts and saves the source node's destination address and PAN ID, and responds with a query response.
4. When the source node receives the query response in `emberIncomingBootloadMessageHandler()`, it checks the target node's EUI, protocol version, and whether the target node is already running the bootloader. The library handles the process of reading the programmed flash pages for the current application image and sends them to the target node.

7.3.7 Bootloader Utility Library API

The bootloader utility library, in /app/util/bootload, provides APIs that source and target node applications can use to interact with a standalone bootloader. The library is supplied as source code.

For details on the bootloader utility, see the library source code and the supplied standalone-bootloader-demo application.

Note: Ember recommends that you do not modify the supplied utilities.

The following sections in this document discuss programming requirements for using the standalone bootloader.

- Library interfaces
- Application requirements
- Bootloader over-the-air launch
- Library constraints

The bootloader utility library contains the following interfaces, defined in bootload utils.h:

Note: Applications that use bootload utilities must define EMBER_APPLICATION_HAS_BOOTLOAD_HANDLERS and EMBER_APPLICATION_HAS_RAW_HANDLERS in their CONFIGURATION_HEADER.

7.3.7.1 Functions

```
void bootloadUtilInit(
   int8u appPort,
   int8u bootloadPort
);

EmberStatus bootloadUtilSendRequest(
   EmberEUI64 targetEui,
   int16u mfgId,
```

```
   int8u hardwareTag[BOOTLOAD_HARDWARE_TAG_SIZE],
   int8u encryptKey[BOOTLOAD_AUTH_COMMON_SIZE],
   int8u mode
);

void bootloadUtilSendQuery(
   EmberEUI64 target
);

void bootloadUtilStartBootload(
   EmberEUI64 target,
   bootloadMode mode
);

void bootloadUtilTick(void);
```

7.3.7.2 Callbacks

```
boolean bootloadUtilLaunchRequestHandler(
   int16u manufacturerId,
   int8u hardwareTag[BOOTLOAD_HARDWARE_TAG_SIZE],
   EmberEUI64 sourceEui
);

void bootloadUtilQueryResponseHandler(
   boolean bootloaderActive,
   int16u manufacturerId,
   int8u hardwareTag[BOOTLOAD_HARDWARE_TAG_SIZE],
   EmberEUI64 targetEui,
   int8u bootloaderCapabilities,
   int8u platform,
   int8u micro,
   int8u phy,
   int16u blVersion
);

#define IS_BOOTLOADING ((blState != BOOTLOAD_STATE_NORMAL) && \
                        (blState != BOOTLOAD_STATE_DONE))
```

7.3.7.3 Application Requirements

To enable over-the-air bootloader launch, network node applications must:

- Include bootload-utils.h in the application's .h file.
- Include the bootload-utils.c file in the project.
- Define `EMBER_APPLICATION_HAS_BOOTLOAD_HANDLERS` and `USE_BOOTLOADER_LIB` in the application's configuration file.
- Implement these handlers:
  - `bootloadUtilLaunchRequestHandler()`
  - `bootloadUtilQueryResponseHandler()`
- Call `bootloadUtilInit()` before `emberNetworkInit()` but after `emberInit()` when the application starts.

> **Note:** If port 1 serves as the bootloader port, bootloadUtilInit()changes the baud rate to 115200.

- Call `bootloadUtilTick()` in a heartbeat function.

The standalone bootloader demonstration application code and libraries allow building different configuration variants depending on what is needed by the application and the code space available. The default is to build using the complete solution.

The following configurations are available when configuring the standalone bootloader demo application and library code. If necessary, you can reduce flash requirements by eliminating features not needed. Typical uses are to remove V1 bootloader support (if not supporting bootloading of legacy Ember devices) or to remove passthrough support.

Modify the following definitions in bootloader-demo-v2-configuration.h:

`USE_BOOTLOADER_LIB` // always defined with the bootloader demo library

`SBL_LIB_SRC_NO_PASSTHRU` // define this to build a node without passthrough ability

`SBL_LIB_TARGET` // define this to build a target only node.

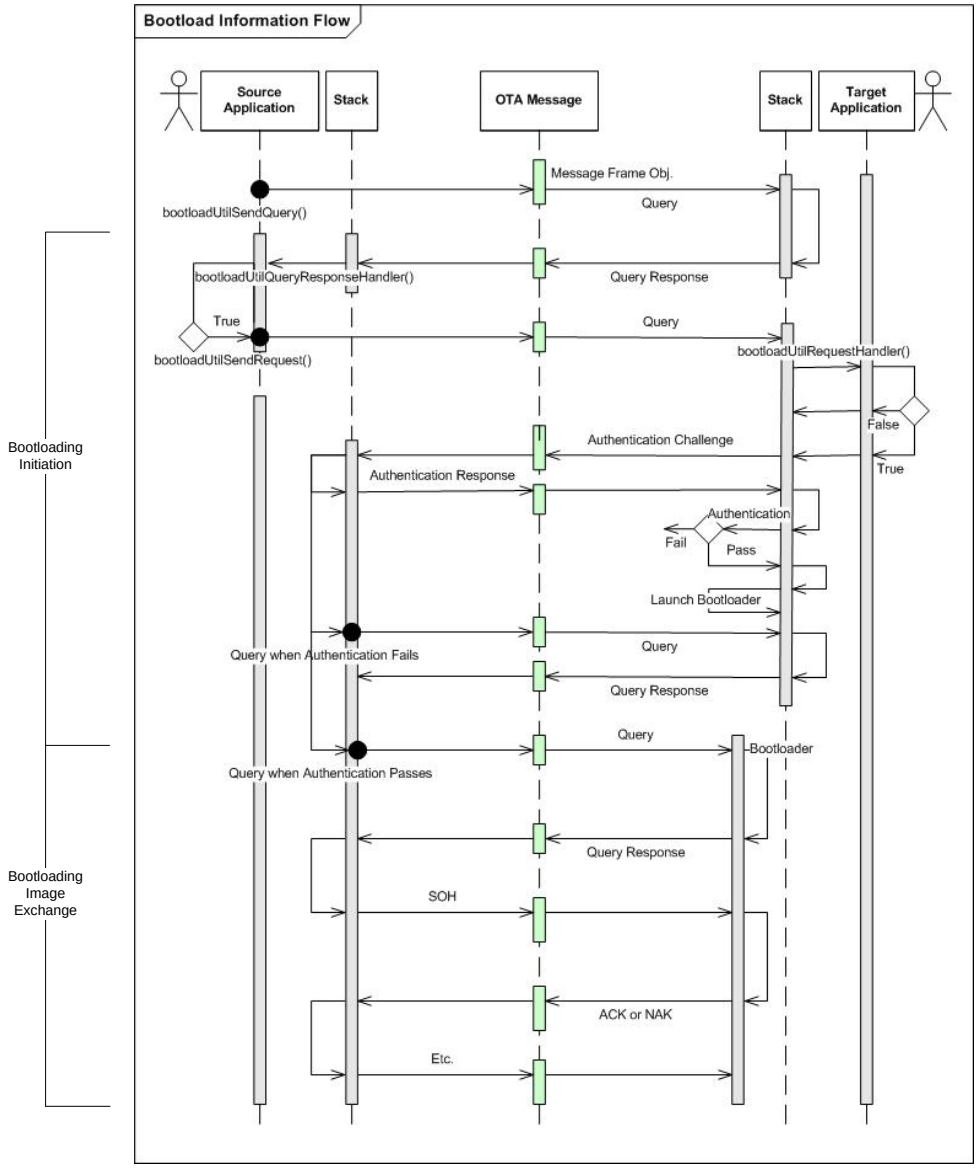
`?EMBER_APPLICATION_HAS_RAW_HANDLER` // configures V1 bootloader protocol support

`EMBER_APPLICATION_HAS_BOOTLOAD_HANDLERS` // leave this defined unless app does not supply incoming message handler. Not defined adds message and transmit complete stubs.

7.3.7.4 Bootloader Over-the-Air Launch

The bootloader utility library in /app/util/bootload provides the implementation of a standard mechanism for over-the-air bootloader launch. This mechanism is compatible with InSight Desktop, and restricts bootloader launch to trusted devices only. This process is summarized in Figure 7-10.

Figure 7-9. Standalone Bootloading Initial Information Flow



Before you can update the image on a device that has an application running, its bootloader must be launched. The process typically follows these steps:

1. The source node typically queries the network to determine which nodes require updating, by issuing an APS message to nodes of interest. Responding nodes return their application version. The source node evaluates this information and identifies potential target nodes accordingly.
2. The source node queries each potential target node by calling `bootloadUtilSendQuery()`. This function can initiate a unicast or broadcast message, depending on the argument supplied—NULL (0xFF) for broadcast, or the target node's EUI for unicast.

3. On each queried node, the application-supplied handler `bootloadUtilQueryResponseHandler()` is invoked, which returns the following information to the source node:
   - Whether it is in application or bootload mode
   - Device type, including platform, micro, phy, and board designations
4. Depending on the query results, the source node can send a bootloader launch request message to a target node by calling `bootloadUtilSendRequest()`.

   You supply the following arguments:
   - Target node's EUI64: Identifies the node to upgrade.
   - Manufacturer's ID: The manufacturer's unique product identifier.
   - Hardware tag: The manufacturer's unique hardware identifier.
   - Encryption key: Manufacturer-supplied, the target node uses this to verify that the source node is authorized to initiate the request.
   - Desired mode: Set to `BOOTLOAD_MODE_PASSTHRU`.
5. On receiving the launch request, the target node calls the bootloader launch handler `bootloadUtilLaunchRequestHandler()`. This handler examines the information supplied by the target node—manufacturer and hardware IDs, radio signal strength, or other metrics—and determines whether the application should allow the request. The handler returns either true (launch the bootloader) or false. If false, the transaction completes when the source node times out waiting for the authorization challenge.
6. If the target device launch handler returns true, the target application calls `bootloadSendAuthChallenge()`, which sends an authorization challenge message to the source device. This challenge contains the target device's EUI64 and random data.
7. When the source device receives the challenge, it calls `bootloadUtilSendAuthResponse()`, which uses the AES block cipher and the encryption key saved from the earlier request, and encrypts the challenge data.

7.3.7.5 Library Constraints

The following constraints apply to the bootloader utility library:

- The library does not support multi-hop downloads.
- The library code takes over the serial port on the source node during serial uploads. If the application uses the serial port, it might need to reconfigure this port when the upload is complete.
- Future releases might require changes to bootloader utility APIs.

7.3.8 Manufacturing Tokens

The bootloader requires you to set several manufacturing tokens. Note that a special area of flash is used to store these tokens, so they cannot be written by an application at runtime. The EM35x uses em3xx_load.exe and the EM250 uses em2xx_patch.exe to set these tokens.

See Ember document 120-5058-000: *Setting Manufacturing Certificates for the EM35x*, as well as Ember documents 120-5031-000: *Bringing Up Custom Devices for the EM250 SoC Platform*, 120-5041-000: *Bringing Up Custom Devices for the EM260 Co-Processor*, and 120-5064-000: *Bringing Up Custom Devices for the EM35x SoC Platform* for more information on setting manufacturing tokens. The tokens that need to be set are:

TOKEN_MFG_BOARD_NAME - Synonymous with the hardware tag used to identify nodes during the bootloader protocol. This tag serves two purposes:

- Applications can query nodes for their hardware tags and can determine which nodes to bootload accordingly.
- When a node calls bootloadUtilSendRequest to request that a target node switch to bootloader mode, it supplies the target's hardware tag as an argument. The target can use this tag to determine whether to refuse to launch its bootloader if it believes the requesting node is trying to program it with software for another hardware type. Each customer is responsible for programming this value.

TOKEN_MFG_MANUF_ID - A 16-bit (2-byte) string that identifies the manufacturer. This tag serves two purposes:

- Applications can query nodes to obtain their manufacturer ID, and decide whether to bootload a node accordingly.
- When a node calls bootloadUtilSendRequest to request that a target node switch to bootloader mode, it supplies the target's manufacturer ID as an argument. The target can refuse to launch its bootloader if it believes the requesting node is trying to program it with software for another manufacturer.

Each customer is responsible for programming this value. Customers are encouraged to use the 16-bit manufacturer's code assigned to their organization by the ZigBee Alliance. This value is typically also used with the EmberZNet PRO stack's emberSetManufacturerCode() API call (stack/include/ember.h) to set the manufacturer ID used as part of the Simple Descriptor by the ZigBee Device Object (ZDO).

TOKEN_MFG_PHY_CONFIG - Configures operation of the alternate transmit path of the radio, which is sometimes required when using a power amplifier. This token should be set as described in the Bringing Up Custom documentation for your platform, or else the bootloader may not operate correctly in a recovery scenario. Each customer is responsible for programming this value.

TOKEN_MFG_BOOTLOAD_AES_KEY - The 16-byte AES key used during the bootloader launch authentication protocol. Each customer is responsible for programming this value and keeping it secret. Ember ships with the AES key set to all 0xFF. The sample application also uses this value. If the value is changed, be sure to modify the application too.

7.3.9 Example Standalone Bootloading Scenario

7.3.9.1 Standalone Bootloader

The standalone bootloader demo shows how to integrate the bootloader utility library into an application. It shows how to trigger all modes of operation, and uses a simple command interface to manually drive the bootloader. You can use the standalone bootloader demo as a development tool or starting point for your own application that will update your device images over the air.

The application features various commands over the serial port to exercise all available bootload features. It does not use any buttons.

7.3.9.2 Serial Baud Rates and Ports Used

The application uses an InSight Adapter feature called virtual UART to communicate with users. All commands are transmitted and received through (TCP) port 4900. Serial port 1 (TCP port 4901) is configured to be the bootload port at 115,200 bps. Table 7-3 lists the serial commands supported.

If you change the settings for serial port 1 in your InSight Adapter's administrative configuration interface, you might need to adjust your configuration to match what this application expects. Refer to Ember document 120-2010-000: *InSight Adapter Technical Specification* for information about adjusting these settings.

Note: Both the application and bootload ports can be configured for the same port. However, application usage of the port needs to be limited when bootloading is in progress in order to maximize performance and to avoid any interruption to the bootload process.

Table 7-3. Serial Commands Supported

Command	Description
`default mode`	Recover nodes that fail bootloading on the default channel (13), where mode is set to 1 (passthrough).
`Form`	Form a network.
`Join`	Join the network as router.
`Leave`	Leave the network.
`query_neighbor`	Report bootload-related information about itself and its neighbor. This information helps determine which node to bootload.
`query_network`	Obtain application information about itself and other nodes in the network.
`recover target-EUI64 mode`	Recover nodes that fail bootloading on current channel, where mode is set to 1 (passthrough).
`remote target-EUI64`	Upload an application image to the specified remote node. The image is obtained from the serial port using XModem protocol.
`serial`	Put the node in serial bootload mode.

7.3.9.3 Usage Notes

1. To start remote bootloading, select the .EBL file to download using a terminal emulator program (such as Hyperterminal) connected to the node's serial port. The node signals the event by printing a stream of C characters to serial port 1.
2. This standalone bootloader demo sample application can be used together with the demo Java application BootloaderDemoV2.java (see section 7.3.9.4, Sample Upload Application).

7.3.9.4 Sample Upload Application

The Ember distribution provides a sample Java application for over-the-air uploads in /tool/standalone-bootloader. This application lets users gather information and perform over-the-air bootloading on network nodes through a gateway node as follows:

1. The application talks to the gateway node over TCP/IP.
2. The gateway node talks to other nodes in the network over-the-air.

This sample application supports the following tasks:

- Querying neighbor nodes to determine which nodes require bootloading.
- Serial bootloading on the gateway node.
- Over-the-air passthrough bootloading on a remote node.
- Recovery and default recovery on a remote node.

7.3.10 OTA Standalone Bootloader Packets

The following sections describe the format of the packets used during over the air bootloading.

7.3.10.1 Broadcast Query

Table 7-4 describes the format of the broadcast query message.

Table 7-4. Broadcast Query Message Format

# bytes	Field	Description/notes
1	Length	Packet length (does not include the length byte)
2	Frame control field	Short destination, long source, inter PAN, command frame
1	Sequence number	
2	Destination PAN ID	Always set to broadcast address 0xFFFF
2	Destination address	Always set to broadcast address 0xFFFF
2	Source PAN ID	
8	Source EUI64	
1	MAC command type	Always set to 0x7C (an invalid 15.4 command frame chosen for bootload packets)
2	Signature	Always set to em; used as further validation in addition to mac command type
1	Version	Version of the bootloader protocol in use, currently set to 0x0001
1	Bootloader command	Always set to 0x51 for query
2	Packet CRC	

7.3.10.2 Common Packet Header

Many of the message packets use a common header format. Table 7-5 describes the format of this common header.

Table 7-5. Common Header for All Other Message Types

# bytes	Field	Description/notes
1	Length	Packet length (does not include the length byte)
2	Frame control field	Long destination, long source, intra PAN, ACK request, command frame
1	Sequence number	A unique identifier for each MAC layer transaction
2	Destination PAN ID	
8	Destination EUI64	
8	Source EUI64	
1	MAC command type	Always set to 0x7C (an invalid 15.4 command frame chosen for bootload packets)
2	Signature	Always set to em; used as further validation in addition to MAC command type
1	Version	Version of the bootloader protocol in use, currently set to 0x0001
n	Data	Remainder of packet

7.3.10.3 Query Packet

Table 7-6. Query Packet

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x51 query
2	Packet CRC	

7.3.10.4 Query Response

Table 7-7. Query Response

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x52 query response
1	Bootloader active	0x01 if the bootloader is currently running; 0x00 if an application is running
2	Manufacturer ID	
16	Hardware tag	
1	Bootloader capabilities	0x00
1	Platform	0x02 xap2b, 0x04 Cortex-M3
1	Micro	0x01 em250, 0x03 em357, 0x05 em351
1	PHY	0x02 em250, 0x03 em3xx
2	blVersion	Optional field. Contains the remote standalone bootloader version. The high byte is the major version; low byte is the build.
2	Packet CRC	

7.3.10.5 Bootloader Launch Request

Table 7-8. Bootloader Launch Request

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x4C launch request
2	Manufacturer ID	
16	Hardware tag	
2	Packet CRC	

7.3.10.6 Bootloader Authorization Challenge

Table 7-9. Bootloader Authorization Challenge

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x63 authorization challenge
16	Challenge data	
2	Packet CRC	

7.3.10.7 Bootloader Authorization Response

Table 7-10. Bootloader Authorization Response

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x72 authorization response
16	Challenge response data	
2	Packet CRC	

7.3.10.8 XModem SOH

Table 7-11. XModem SOH

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x01 XModem SOH
1	Block number	
1	Block number one's complement	
64	Data	
2	Block CRC	
2	Packet CRC	

7.3.10.9 XModem EOT

Table 7-12. XModem EOT

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x04 XModem EOT
2	Packet CRC	

7.3.10.10 XModem ACK

Table 7-13. XModem ACK

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x06 XModem ACK
1	Block number	
2	Packet CRC	

7.3.10.11 XModem NACK

Table 7-14. XModem NACK

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x15 XModem NACK
1	Block number	
2	Packet CRC	

7.3.10.12 XModem Cancel

Table 7-15. XModem Cancel

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x18 or 0x03 XModem cancel (from source)
2	Packet CRC	

7.3.10.13 XModem Ready

Table 7-16. XModem Ready

# bytes	Field	Description/notes
26	Common header format	
1	Bootloader command	0x43 XModem ready
2	Packet CRC	

7.3.11 Ember Bootload (EBL) File Format

The Ember Bootload (EBL) format has two main components:

- Header field
- Tag-separated data fields

The file image data is separated into segments delineated by tags in TLV (tag, length, value) format. The tag describes the associated value and the operation on it. The tag and length fields are each 2 bytes. The value or data is length bytes long.

The header file defines four tag types (located in hal/micro/cortexm3/bootloader/ebl.h for EM35x and hal/micro/xap2b/em250/em250-ebl.h):

```
#define EBLTAG_HEADER      0x0000
#define EBLTAG_PROG        0xFE01
#define EBLTAG_ERASEPROG   0xFD03
#define EBLTAG_END         0xFC04
```

7.3.11.1 Header

The main difference between EM35x and EM250 EBL file formats is the header. The EM250 header is 64 bytes and all of the fields are saved/sent in big endian order (MSB first). The following is a copy of the structure for easy reference, but refer to em250-ebl.h for the official definition.

```
typedef struct      eblHdr_s
      {
        int16u      tag;            /* = EBLTAG_HEADER (always 0x0000) */
        int16u      len;            /* = 60 (always 0x003C) */
        int16u      flashAddr;      /* Location in flash of header     */
        int16u      signature;      /* .ebl signature for EM250 (always
0xE250)*/
        int32u      timestamp;      /* Unix epoch time of .ebl file    */
        int32u      imageCrc;       /* CRC over following pageRanges   */
        pageRange_t pageRanges[6];  /* Flash pages used by app         */
        int8u       imageInfo[32];  /* PLAT-MICRO-PHY-BOARD string     */
        int32u      headerCrc;      /* Header CRC   Little-Endian-     */
      }                   eblHdr_t;
      typedef struct      pageRange
      {
         int8u      begPg;          /* First flash page in range      */
```

```
        int8u      endPg;          /* Last  flash page in range
(inclusive   */
  }         pageRange_t;
```

On the EM35x, some of the fields that used to be in the EBL header were moved into an Application Address Table (AAT) structure within the header. Below are copies of the definitions of the EBL header and the AAT for you convenience. Please refer to `ebl.h` and `hal/micro/cortexm3/memmap-tables.h` for the latest definitions.

```
typedef struct eblHdr3xx_s
{
  int16u      tag;         /* = EBLTAG_HEADER             */
  int16u      len;         /* =                           */
  int16u      version;     /* Version of the ebl format   */
  int16u      signature;   /* Magic signature: 0xE350     */
  int32u      flashAddr;   /* Address where the AAT is stored */
  int32u      aatCrc;      /* CRC of the ebl header portion of the AAT */
  // aatBuff is oversized to account for the potential of the AAT to grow
  // in the future.  Only the first 128 bytes of the AAT can be
  // referenced as part of the ebl header, although the AAT itself may
  // grow to 256 total
  int8u       aatBuff[128]; /* buffer for the ebl portion of the AAT */
} eblHdr3xx_t;

typedef struct {
  HalBaseAddressTableType baseTable;
  // The following fields are used for ebl and bootloader processing.
  //   See the above description for more information.
  int8u platInfo;   // type of platform, defined in micro.h
  int8u microInfo;  // type of micro, defined in micro.h
  int8u phyInfo;    // type of phy, defined in micro.h
  int8u aatSize;    // size of the AAT itself
  int16u softwareVersion;  // EmberZNet SOFTWARE_VERSION
  int16u reserved;  // reserved 16 bits of padding
  int32u timestamp; // Unix epoch time of .ebl file, filled in by ebl gen
  int8u imageInfo[IMAGE_INFO_MAXLEN];  // string, filled in by ebl gen
  int32u imageCrc; // CRC over following pageRanges, filled in by ebl gen
  pageRange_t pageRanges[6];  // Flash pages used by app, filled in by
ebl gen

  void *simeeBottom;

  // reserve the remainder of the first 128 bytes of the AAT in case we
  // need to go back and add any values that the bootloader will need to
  // reference, since only the first 128 bytes of the AAT become part of
  // the EBL header.
  int32u bootloaderReserved[10];

  //////////////
  // Any values after this point are still part of the AAT, but will not
  //   be included as part of the ebl header

  void *debugChannelBottom;
  void *noInitBottom;
  void *appRamTop;
  void *globalTop;
  void *cstackTop;
  void *initcTop;
  void *codeTop;
```

```
  void *cstackBottom;
  void *heapTop;
} HalAppAddressTableType;
```

An .EBL file has only one EBLTAG_HEADER tag, and it must be the first tag in the file. The value of 0x0000 for the header tag is intentional, chosen so the older version 1 .EBL code will not process a version 2 file. Version 1 requires that the length be in the first two bytes, and a length of zero would be invalid.

7.3.11.2 Program Data

The two program tags are EBLTAG_PROG and EBLTAG_ERASEPROG. The data following the tag starts with a two-byte address field. If the tag was an EBLTAG_ERASEPROG then the flash page containing the tag's address is erased before the data is written. For an EBLTAG_PROG tag, the data is written to that address without an erase cycle.

Flash is erased in 2 kB segments on the EM35x (1 kB [512-word] segments on the EM250), so the .EBL file should only have one EBLTAG_ERASEPROG tag for each 2 kB (1 kB on EM250) chunk. After the initial erase any other writes to that page should be done using EBLTAG_PROG tags so as not to erase the newly written data.

The following is the EBLTAG_PROG and EBLTAG_ERASEPROG tag definition from ebl.h (em250-ebl.h on the EM250):

```
typedef struct      eblProg
{
   int16u      tag;         /* = EBLTAG_[ERASE]PROG     */
   int16u      len;         /* = 2..65534               */
   int16u      flashAddr;   /* Starting addr in flash   */
   int16u      flashData[2/*len/2*/];
}         eblProg_t;
```

7.3.11.3 End of File Indication

The end of a file is indicated with an `EBLTAG_END` tag. Its data field contains a 4-byte CRC of the whole download file, including the tags but excluding the header. This CRC can be used to verify the integrity of the download file stream. If the bootloader consuming this download file keeps a running CRC on each byte received, the running CRC can be compared to the value received in this tag.

The CRC in the tag is the one's complement of the running CRC, and it is stored and sent in LITTLE_ENDIAN order (LSB first). This way, it can be included in the running CRC of the receiver, and if it is correct, results in the CRC algorithm's magic remainder of 0xDEBB20E3.

The following is the `EBLTAG_END` tag definition from ebl.h (em250-ebl.h on the EM250):

```
typedef struct      eblEnd
{
   int16u      tag;         /* = EBLTAG_END         */
   int16u      len;         /* = 4                  */
   int32u      eblCrc;      /* CRC of .ebl image    */
}         eblEnd_t;
```

7.3.11.4 Data Verification

The EBL file format includes three 32bit CRC values to verify the integrity of the file. These values are computed using the `halCommonCrc32()` function which can be found in `hal/micro/generic/crc.c`. The initial value of the CRC used in the computation is 0xFFFFFFFF.

Table 7-17 describes the data integrity checks built into the .EBL download format.

Table 7-17. EBL Data Integrity Checks

Integrity Check	Description
Header CRC	The header data contains the headerCrc field (aatCrc on the EM35x), a 4 byte, one's complement, LSB-first CRC of the header bytes only. This is used to verify the integrity of the header. On the EM35x this CRC assumes that the value of the type field in the AAT is set to 0xFFFF.
EBLTAG_END CRC	The end tag value is the one's complement, LSB-first CRC of the data download stream, including the header and ending at the end tag. This is used as a running CRC of the download stream, and it verifies that the download file was received properly.
Image CRC	The header's imageCrc field is the one's complement, MSB-first CRC of all the flash pages to be written including any unused space in the page (initialized to 0xFF). It does not include the EBL tag data and assumes that the first 128 bytes of the AAT on the EM35x and the whole EBL Header on the EM2xx are set to 0xFF. This is used after the image download is complete and everything but the header has been written to flash to verify the download. The download program does this by reading each flash page written as it is defined in the header's pageRanges[] array and calculating a running CRC.

7.4 Application Bootloading

Application bootloading is very similar on the EM35x and EM250. The concept of application bootloading is not compatible with the EM260. The EM260 can be upgraded over the air using the standalone bootloader or using the SPI or UART interface and having the host processor complete the bootloading operation.

7.4.1 Introduction

The application bootloader has the single purpose of reprogramming the flash with an application image stored in some download space. Typically this download space is an external memory device like a dataflash or EEPROM. The actual download of the new image is handled by the application, which makes the actual bootloader much simpler than the standalone bootloader. One other piece of code used with the application bootloader is the recovery image. The recovery image uses the serial port to load a new application image when the application cannot be run for any reason.

By separating the various steps of upgrading a device's application, the application bootloader provides maximum flexibility. The application is free to upload the new image over-the-air from multiple hops away if necessary, over a time duration that fits with the application, such as all at once or slowly over time.

By storing the newly uploaded image externally, the application doesn't need to be overwritten until the new image has been successfully saved. Also, a node can save an image and, later, forward that image to other nodes. If extra download space is available, a node could potentially store various images for other nodes, or multiple versions for the node in question.

This flexibility does come at some cost - mostly in the form of adding another device to the design in which to store the image.

A bootloader library is supplied to assist you with the task of interfacing the application code to the bootloader function. This library allows the application to read and write the external memory, to verify the uploaded image is intact, and to reset the module to allow the bootloader to flash the new image.

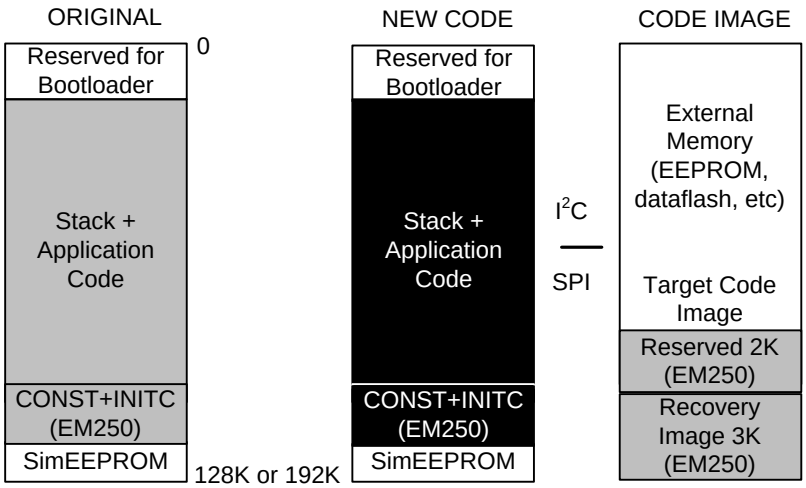
7.4.2 Memory Map

The EM250 application bootloader uses the same memory map as the EM250 standalone bootloader, with two exceptions. The bootloader segment is smaller, 5 kB as compared with 10 kB, and it uses an external memory, which is addressed separately.

The EM35x uses the same 8 kB memory map for the application bootloader as it does for the standalone bootloader, but with an external memory that is addressed separately.

For the EM250 a recovery image is stored in the top 3 kB of the external memory, while this image is built into the EM35x. A diagram of the memory layout is shown in Figure 7-11.

Figure 7-10. Application Bootloading Code Space (Typical)



7.4.3 Remote Memory Connection

The application bootloader uses a remote device to store the downloaded application image.

The application bootloader can access its remote storage using either an I2C or SPI interface. The EM250 driver for the I2C interface supports the Microchip 24AA1025 128 kB Serial EEPROM. The driver for the SPI interface supports the Atmel AT45DB011B 132 kB Serial Flash device. The Atmel AT45DB011B is a 1-megabit DataFlash organized as 512

pages of 264 bytes each. The application bootloader utilizes 256 bytes per page for a total of 128 kBs, the same size as the EM250 flash.

The EM35x release includes sample drivers for the Atmel AT45DB021D 2 Mb SPI DataFlash and the Microchip 24AA1025 1 Mb I2C EEPROM. The 24AA1025, at 1 Mb (128 kB x 8), does not completely cover the EM35x's 192 kB flash size.

Both the I2C and SPI versions of the application bootloader are supplied, in .xpv/.xdv format for the EM250 and S-record format for the EM35x. On the EM35x, in addition to these prebuilt images, you can build the bootloader image yourself using the supplied IAR Embedded Workbench project file (found under em35x/tool/bootloader-em35(1|7)/app-bootloader/app-bootloader.eww). By modifying the project file and rebuilding the bootloader you can do things like use a custom remote storage driver or setup your own default GPIO configuration.

The I2C bus is configured to use the SC2 Serial Controller at 400 kbps in master mode.

The SPI bus is configured to use the SC2 Serial Controller with the following settings: 12 MHz, master mode, MSB transmitted first, sample on leading edge, rising leading edge.

Schematics for connecting a SPI based dataflash chip to an EM35x can be found in the hardware reference designs distributed with the stack.

7.4.3.1 Remote Memory Access

The file bootloader-interface-app.c contains access routines for the external memory device. For more information about these functions see the bootloader-interface-app.h header file.

7.4.4 Loading

7.4.4.1 Application Bootloader

The application bootloader image can be loaded into the device using the em3xx_load.exe or em2xx_load.exe utility. The format of the load command is:

For EM35x: `em3xx_load.exe app-bootloader.s37 --ip <device_ip_addr>`

For EM250: `em2xx_load.exe app-bootloader -Reset -id <device_ip_addr>`

7.4.4.2 Recovery Image

The EM35x application bootloader contains the recovery image so there is no need to store it in the external memory.

The EM250 recovery image can be loaded into the EM250 using the em2xx_load.exe utility. Because it only uses RAM, the -ramCodeOnly option must be used. The two formats of the load command are:

```
em2xx_load.exe -ramCodeOnly em250-app-bootloader-i2c-recovery-ram -Run -
id <device_ip_addr>
```

```
em2xx_load.exe -ramCodeOnly em250-app-bootloader-spi-recovery-ram -Run -
id <device_ip_addr>
```

Executing this command loads the recovery image into the EM250 RAM through the InSight Adapter. Once loaded, the recovery image is run. The recovery image first checks to see if it has been copied into the download space and if not, it copies itself there.

To avoid loading the recovery image at manufacturing time with em2xx_load.exe, the EEPROM or dataflash can be programmed prior to module assembly.

7.4.5 Acquiring a new Image

The application bootloader relies on application code or the recovery image to obtain new code images. The bootloader itself only knows how to read an EBL image stored in the external memory download space and copy the relevant portions to the main flash block. This approach means that the application developer is free to acquire the new code image in any way that makes sense for them (serial, OTA, etc).

Typically application developers choose to acquire the new code image over-the-air since this is readily available on all devices. For OTA bootloading Ember recommends using the standard Zigbee OTA Bootload Cluster. Code for this cluster is available in Ember's Application Framework as a plugin. You can find documentation of this plugin and the protocol in Chapter 15 of 120-3028-000D: *Application Framework V2 Developer Guide*.

7.4.6 Recovery Image

The recovery image is used as a failsafe mechanism to recover a module without a valid application image. It is invoked by the application bootloader when the local flash image and the image in the external memory are invalid or by manually entering the Emergency Recovery mode. Its only function is to upload a new application over a serial line.

On the EM35x platform, the recovery image is built into the application bootloader itself and is launched as a separate mode of operation. It can be activated by grounding the PA5 GPIO, then sending a carriage return at 115200 baud. The EM35x platform also can be configured to use other IO pins or other schemes of activation by modifying the `bootloadForceActivation()` API in bootloader-gpio.c and rebuilding the bootloader. An example of utilizing PC6, which is connected to a button on the EM35x Breakout Board, is provided in bootloader-gpio.c and can be enabled by building the bootloader with `USE_BUTTON_RECOVERY` defined.

On the EM250, the recovery image is completely separate from the application bootloader and resides in a private 3 kB segment of the external memory. The recovery image is typically loaded onto the device at manufacturing time and is not over-the-air upgradeable in the field. The device can be forced to run the recovery image by powering the device while grounding GPIO 5. This would be used when an application is unresponsive.

The recovery image on the EM250 is unusual in that it only uses system RAM. It has no flash component. When needed, the application bootloader copies the recovery image from the download space to RAM. It then jumps to the start of code and runs from there. This is faster than flashing the image and helps to conserve the lifetime of the

flash memory by not erasing and writing it whenever the recovery image needs to run. This method also provides more flash space for application code.

7.4.6.1 Running the Recovery Image

The recovery image is only run in two instances on the EM35x. The first is when the application bootloader invokes it; and the second is when Emergency Recovery mode is initiated.

On the EM250 the recovery image is run in one additional situation and that is at manufacturing time when the em2xx_load program initially loads it.

7.4.6.2 Functionality in Recovery Image

Once the recovery image has been loaded to RAM and run, it executes as a captive program. Its only task is to upload an image over the serial line using the XModem protocol. On the EM250, as previously mentioned, it first verifies that it has been copied to the external memory.

The recovery image immediately starts the XModem upload sequence by sending 'C' characters out the serial line. The SC1 serial controller is used as a UART at 115,200 baud, 8 bits, no parity, 1 stop bit. The 'C' characters are sent every 1 second until an upload sequence is detected.

Use a terminal emulator on a PC to send the application EBL file to the node connected by serial cable.

Once the image has been saved to the download space, the recovery image resets the module to come up in the bootloader for image processing.

7.4.6.3 Output of Recovery Image

No command line interface is associated with the recovery image. Its only output is on the serial line connected to the terminal emulator. When it initially writes itself to the download space it emits a 'W' character for every 128-byte page it writes. After successfully writing the image to the download space or verifying that the image has previously been saved there, it emits 'OK'. Once the image is verified, it emits 'C's every second waiting for upload.

7.4.7 Errors During Application Bootloading

7.4.7.1 Application Bootloader Errors

The application and the bootloader have limited indirect contact. Their only interaction is through passing non-volatile data across module reboots.

Once the application decides to install a new image saved in the download space it calls `halAppBootloaderInstallNewImage()`. This call sets the bootload mode and reboots the module. If the bootloader fails to install the new image it sets the reset cause to `RESET_BOOTLOADER_IMG_BAD` and resets the module. Upon startup, the application should read the reset cause with `halGetResetInfo()`. If the reset cause is set to `RESET_BOOTLOADER_IMG_BAD` the application knows the install process failed and can attempt to download a new image. A printable error string can be acquired from calling `halGetResetString()`. The application bootloader does not print out on the serial line.

7.4.7.2 Recovery Image Errors

If the recovery image encounters an error while uploading an image, it prints "Err" or "Stat" on the serial line followed by the error or status number in hex. It then restarts the upload process, emitting 'C' characters while waiting for a new upload to start. Table 7-18 lists possible bootload errors.

Table 7-18. Bootload Errors

Error/Status	Description
16	Timeout: Exceeded 60 seconds serial download timeout
18	File abort: Control-C on console
83	Write check error: Data read from external storage does not match data written
84	Image Size error: Download image size is greater than external storage space available

8 The Token System

A token is an abstract data constant that has special persistent meaning for an application. This chapter describes tokens and shows how to use them in code. A token can be one of several types:

- Standard
- Indexed
- Counter
- Manufacturing

Tokens can be categorized in the following general groups:

- Stack tokens – these tokens are read/write and defined in every application to support stack behavior. These tokens live in Simulated EEPROM.
- Application tokens – these tokens are read/write and defined by the application to support application behavior. It is left open to a customer to decide if there are application tokens, how they are defined, and what they do. These tokens live in Simulated EEPROM.
- Fixed manufacturing tokens – these tokens are non-modifiable read only tokens specific to each individual chip and set during chip production. They support key low-level and high-level functionality.
- Customer manufacturing tokens – these tokens are read only from on chip code and read/write from external programming tools. Some of these tokens have an Ember defined purposes to support stack behavior. It is left open to a customer to decide if there are any additional manufacturing tokens, how they are defined, and what they do.

The chapter also discusses bindings, the application-defined associations between two devices on a network.

8.1 Introduction

A token has two parts: a **token key** and **token data**. The token key is a unique identifier that is used to store and retrieve the token data. In many cases, the word "token" is used quite loosely to mean the token key, the token data, or the combination of key and data. Usually it is clear from the context which meaning to use. In this document **token** always refers to the key + data pair.

Tokens are typically stored in NVRAM which is most commonly directly in flash (manufacturing tokens) or in the Simulated EEPROM (stack and application tokens) which operates on top of flash.

8.1.1 Purpose

The fundamental purpose of the token system as compared to generic RAM usage is to allow the token data to persist across reboots and during power loss. By using the token key to identify the proper data, the application requesting the token data does not need to know the exact storage location of the data. This simplifies application design and code reuse. Tokens are also useful when the underlying storage of the data may change over time across implementations.

The EM2xx and EM3xx chips use a special memory-rotation algorithm, called Simulated EEPROM, to prevent premature overuse of the underlying flash. Ember software reserves a section of flash for use by the Simulated EEPROM. EM2xx chips reserve 8 kB of flash. EM3xx chips have the choice of two Simulated EEPROM sizes: 4 kB, 8 kB (the default). The Simulated EEPROM storage area implements a wear-leveling algorithm to extend the number of Simulated EEPROM write cycles beyond the physical limit of the flash. Ember recommends that application designers familiarize themselves with the Simulated EEPROM and its limitations, so that they design the application's use of tokens for optimal flash write cycles. Refer to Ember document 120-5037-000, *Using the Simulated EEPROM,* for more information. However, application designers do not need to know any of the underlying details. Understanding how to use token keys is enough.

8.1.2 Usage

EmberZNet PRO provides a simple set of APIs for accessing token data. The full documentation may be found in EmberZNet PRO API reference for your platform (120-3022-000 for the EM35x, 120-3016-000 for the EM250, ad 120-3020 for the EM260).

The basic API functions include:

```
void halCommonGetToken( data, token )
void halCommonSetToken( token, data )
```

In this case, 'token' is the token key, and 'data' is the token data.

Two special types of tokens are available: **indexed tokens** and **counter tokens**. An indexed token can be used when the data to be stored is an array in which each element may be accessed and updated independently from the others. A counter token can be used when the default operation on the token is to retrieve it, increment it by one, and then store it again.

The API functions for these two types of tokens are:

```
void halCommonGetIndexedToken( data, token, index )
void halCommonSetIndexedToken( token, index, data )
void halCommonIncrementCounterToken( token )
```

The counter token can also be accessed with the normal `halCommonGetToken()`, `halCommonSetToken()` calls. Setting a counter token severely degrades Simulated EEPROM performance, and therefore should only be done when absolutely necessary.

8.2 Accessing Standard (Non-indexed) Tokens

Some applications may need to store configuration data at installation time. Usually, this is a good application for a standard token. Assume the token is defined to use the token key `DEVICE_INSTALL_DATA`, and the data structure looks like this:

```
typedef struct {
  int8u install_date[11] /** YYYY-mm-dd + NULL */
  int8u room_number; /** The room where this device is installed */
} InstallationData_t;
```

Then you can access it with a code snippet like this:

```
InstallationData_t data;
// Read the stored token data
halCommonGetToken(&data, TOKEN_DEVICE_INSTALL_DATA);
// Set the local copy of the data to new values
data.room_number = < user input data >
MEMCOPY(data.install_date, < user input data>, 0,
sizeof(data.install_date));
// Update the stored token data with the new values
halCommonSetToken(TOKEN_DEVICE_INSTALL_DATA, &data);
```

Instructions for configuring custom tokens are provided in Section 8.6, Custom Tokens.

8.3 Accessing Indexed Tokens

To store a set of similar values, such as an array of preferred temperature settings throughout the day, use the default data type `int16s` to store the desired temperatures, and define a token called `HOURLY_TEMPERATURES`.

A local copy of the entire data set would look like this:

```
int16s hourlyTemperatures[HOURS_IN_DAY]; /** 24 hours per day */
```

In the application code, you can access or update just one of the values in the day using the indexed token functions:

```
int16s getCurrentTargetTemperature(int8u hour) {
  int16s temperatureThisHour = 0; /** Stores the temperature for return
*/
  if (hour < HOURS_IN_DAY) {
    halCommonGetIndexedToken(&temperatureThisHour,
TOKEN_HOURLY_TEMPERATURES, hour);
  }
  return temperatureThisHour;
}
void setTargetTemperature(int8u hour, int16s targetTemperature) {
  if (hour < HOURS_IN_DAY) {
    halCommonSetIndexedToken(TOKEN_HOURLY_TEMPERATURE, hour,
&temperatureThisHour);
  }
}
```

8.4 Accessing Counter Tokens

Counting the number of heating cycles a thermostat has initiated is a perfect use for a counter token. Assume it is named `LIFETIME_HEAT_CYCLES`, and it is an `int32u.`

```
void requestHeatCycle(void) {
  /// < application logic to initiate heat cycle >
  halCommonIncrementCounterToken(TOKEN_LIFETIME_HEAT_CYCLES);
}
int32u totalHeatCycles(void) {
  int32u heatCycles;
  halCommonGetToken(&heatCycles, TOKEN_LIFETIME_HEAT_CYCLES);
  return heatCycles;
}
```

8.5 Accessing Manufacturing Tokens

Manufacturing tokens are defined and treated as standard (non-indexed) tokens. The major difference is manufacturing tokens can only be read from on chip code and customer manufacturing tokens can only be written with external programming tools. In addition to being able to read manufacturing tokens with the usual standard token read, `halCommonGetToken()`, manufacturing tokens can also be read with their own dedicated API, `halCommonGetMfgToken()`. This MfgToken API takes the same parameters at the standard GetToken API. In general, MFG tokens should be accessed through the standard GetToken API. The two primary purposes for using the MfgToken API is for slightly faster access and early on in the boot process before `emberInit()` is called.

8.6 Custom Tokens

Custom application tokens are defined in a header file. The header file can be specific to each project, and is defined by the preprocessor variable `APPLICATION_TOKEN_HEADER`. Custom manufacturing tokens require editing an existing header file. For the EM35x, this header file is located at hal/micro/cortexm3/token-manufacturing.h. For the EM250, this header is located at hal/micro/avr-atmega/token-manufacturing.h. Defining custom manufacturing tokens is generally considered a rare practice, but is still possible. Note that to define them requires editing a file that contains key definitions critical to the system so making any additions should be done very carefully and should follow the same format as the other definitions in the file.

The `APPLICATION_TOKEN_HEADER` file should have the following structure:

```
/**
* Custom Application Tokens
*/
// Define token names here
#ifdef DEFINETYPES
// Include or define any typedef for tokens here
#endif //DEFINETYPES
#ifdef DEFINETOKENS
// Define the actual token storage information here
#endif //DEFINETOKENS
```

Note: The header files do not have #ifndef HEADER_FILE / #define HEADER_FILE sequence at the top. This is important because this header file is included several times for different purposes.

8.6.1 Mechanics

Adding a custom token to the header file involves three steps:

1. Define the token name.
2. Add any typedef needed for the token, if it is using an application-defined type.
3. Define the token storage.

The following illustrates how to define the three previous token examples.

8.6.1.1 Define the Token Name

When defining the name, do not prepend the word TOKEN. Instead, use the word CREATOR:

```
/**
* Custom Application Tokens
*/
// Define token names here
#define CREATOR_DEVICE_INSTALL_DATA (0x000A)
#define CREATOR_HOURLY_TEMPERATURES (0x000B)
#define CREATOR_LIFETIME_HEAT_CYCLES (0x000C)
```

This defines the token key and links it to a programmatic variable. The token names are actually `DEVICE_INSTALL_DATA, HOURLY_TEMPERATURES,` and `LIFETIME_HEAT_CYCLES`, with different tags prepended to the beginning depending on the usage. Thus they are referred to in the example code as `TOKEN_DEVICE_INSTALL_DATA`, and so on.

The token key is a 16-bit value that must be unique within this device. The first-bit is reserved for manufacturing and stack tokens, so all custom tokens should have a token key less than 0x8000.

The token key is critical to linking application usage with the proper data and as such a unique key should always be used when defining a new token or even changing the structure of an existing token. Always using a unique key guarantees a proper link between application and data.

8.6.1.2 Define the Token Type

Each token in this example is a different type; however, the `HOURLY_TEMPERATURES` and `LIFETIME_HEAT_CYCLES` types are built-in types in C. Only the `DEVICE_INSTALL_DATA` type is a custom data structure.

The token type is defined using the structure introduced at the start of section 8.6, Custom Tokens, and repeated below. Note that the token type must be defined only in one place, as the compiler will complain if the same data structure is defined twice.

```
#ifdef DEFINETYPES
// Include or define any typedef for tokens here
typedef struct {
```

```
  int8u install_date[11] /** YYYY-mm-dd + NULL */
  int8u room_number; /** The room where this device is installed */
} InstallationData_t;
#endif //DEFINETYPES
```

8.6.1.3 Define the Token Storage

After any custom types are defined, the token storage is defined. This informs the token management software about the tokens being defined. Each token, whether custom or built-in, gets its own entry in this part:

```
#ifdef DEFINETOKENS
// Define the actual token storage information here
DEFINE_BASIC_TOKEN(DEVICE_INSTALL_DATA,
  InstallationData_t,
  {0, {0,...}})
DEFINE_INDEXED_TOKEN(HOURLY_TEMPERATURES, int16u, HOURS_IN_DAY, {0,...})
DEFINE_COUNTER_TOKEN(LIFETIME_HEAT_CYCLES, int32u, 0}
#endif //DEFINETOKENS
```

The following expands on each step in this process.

```
DEFINE_BASIC_TOKEN(DEVICE_INSTALL_DATA,
  InstallationData_t,
  {0, {0,...}})
```

`DEFINE_BASIC_TOKEN` takes three arguments: the name (`DEVICE_INSTALL_DATA`), the data type (`InstallationData_t`), and the default value of the token if it has never been written by the application (`{0, {0,...}}`).

The default value takes the same syntax as C default initializers. In this case, the first value (`room_number`) is initialized to 0, and the next value (`installation_date`) is set to all 0s because the `{0,...}` syntax fills the remainder of the array with 0.

The syntax of `DEFINE_COUNTER_TOKEN` is identical to `DEFINE_BASIC_TOKEN`.

`DEFINE_INDEXED_TOKEN` requires a length of the array -- in this case, `HOURS_IN_DAY`, or 24. Its final argument is the default value of every element in the array. Again, in this case it is initialized to all 0.

The next section discusses default tokens, including stack tokens and manufacturing tokens.

8.6.2 Default Tokens

The EmberZNet PRO stack contains some default tokens that may be useful for the application developer. These tokens come in two flavors:

Stack Tokens are runtime configuration options set by the stack; these should not be changed by the application.

Manufacturing Tokens are set at manufacturing time and cannot be changed by the application.

To view the stack tokens, refer to the file:

<install-dir>/stack/config/token-stack.h

To view the manufacturing tokens for your chip, refer to one of the follow files:

<install-dir>/hal/micro/xap2b/token-manufacturing.h

<install-dir>/hal/micro/cortexm3/token-manufacturing.h

Search for `CREATOR` to see the defined names. If the entire file seems overwhelming, focus only on the section describing the tokens.

Some of the fixed manufacturing tokens may be set by the manufacturer when the board is created. For example, a custom EUI-64 address may be set by the vendor to override the internal EUI-64 address provided by Ember. Other tokens, such as the internal EUI-64, cannot be overwritten.

For more information about manufacturing and token programming, refer to Ember documents 120-5031-000: *Bringing Up Custom Devices for the EM250 SoC Platform*, 120-5041-000: *Bringing Up Custom Devices for the EM260 Co-Processor*, and 120-5064-000: *Bringing Up Custom Devices for the EM35x SoC Platform*.

8.7 Bindings

Bindings are application-defined associations between two devices on the network. A binding may be used when sending to any message destination (multicast group or unicast point). They are used to provide a persistent means to store a pairing of two devices and the clusters, endpoints, and destinations they are using. The binding table consumes 2 bytes of RAM and 13 bytes of flash for each entry. Parts of this table reside in RAM, but most of the data is stored in token data. Therefore, the number of available bindings is limited by the processor's available RAM and EEPROM. It is important to understand that your application is responsible for managing binding table entries. For detailed information on `EMBER_BINDING_TABLE_SIZE` and `TOKEN_BINDING_TABLE`, refer to the EmberZNet PRO API reference for your platform (120-3022-000 for the EM35x, 120-3016-000 for the EM250, and 120-3020 for the EM260).

The `EMBER_BINDING_TABLE_SIZE` constant specifies the maximum number of bindings supported by the stack. This includes the bindings in EEPROM and in RAM. It is set in stack/include/ember-configuration-defaults.h, and can be reset through preprocessor definitions at compile time.

8.8 For More Information

For more detailed information about the underlying token mechanism and the simulated EEPROM that provides the EM2xx and EM3xx NVRAM implementation, please refer to the EmberZNet PRO API reference for your platform (120-3022-000 for the EM35x, 120-3016-000 for the EM250, and 120-3020 for the EM260), which has detailed information about the use of these items.

9 Application Development Tools

9.1 Introduction

As with most embedded development technologies, Ember provides a set of tools to allow you (the developer) to create a product using Ember's ZigBee products. Each Ember chip family has a toolchain associated with it that addresses its unique development requirements. Wherever possible, we have selected the best development tools available, or we have developed the tool on our own.

This chapter provides an overview of the toolchain that you will use to develop, build and deploy your applications. These tools fall into one of three categories:

- EmberZNet PRO Stack Software
- Compiler Toolchain
- Application Development and Debugging Toolchain

The actual toolchain that you will use is device and processor model-dependent. For this discussion, processor model is either System-on-Chip (SoC) or Network Coprocessor (NCP). The SoC model requires that the customer application to be co-resident with Ember's stack. The NCP model requires that the customer application be on a separate host processor and the ZigBee stack run on the NCP. Table 9-1 summarizes the major tools for each device.

Table 9-1. Toolchain Summary

EmberZNet PRO Stack Software	Compiler	Application Development and Debugging
EM35x SoC		
Stack Libraries, HAL source, API Documentation, Sample Applications, Optional Development Kit	IAR EWARM: IDE: Compiler, Online Help; Debugger (device level); Document Library	InSight Desktop + AppBuilder + Online Help
EM250 SoC		
Stack Libraries, HAL source, API Documentation, Sample Applications, Optional Development Kit	xIDE Compiler:IDE+Online Help; Debugger (device level); Document Library; Template Projects	InSight Desktop + AppBuilder + Online Help
EM260 / EM35x NCP		
Stack Libraries, HAL source, API Documentation, Sample Applications, Utilities, Optional Development Kit	3rd Party Compiler Toolchain (depends upon Host Processor Selection)	InSight Desktop + AppBuilder + Online Help

Misc Tools & Utilities

In addition to the major tools above, Ember also supplies a number of single function tools and utilities such as

- Bootloaders
- Programming Support Tools
- Token Utility

Ember also sells a variety of development kit hardware to suit various needs.

The following sections provide more detail about the most important elements of the toolchain.

9.2 EmberZNet Stack Software

The EmberZNet Stack Software is a collection of libraries, source code, tools, sample applications, and product documentation. The latest version of EmberZNet at the time of this writing is release 4.x, also known as EmberZNet PRO. Starting with EmberZNet 4.2, all Ember chip families are supported on the same stack release.

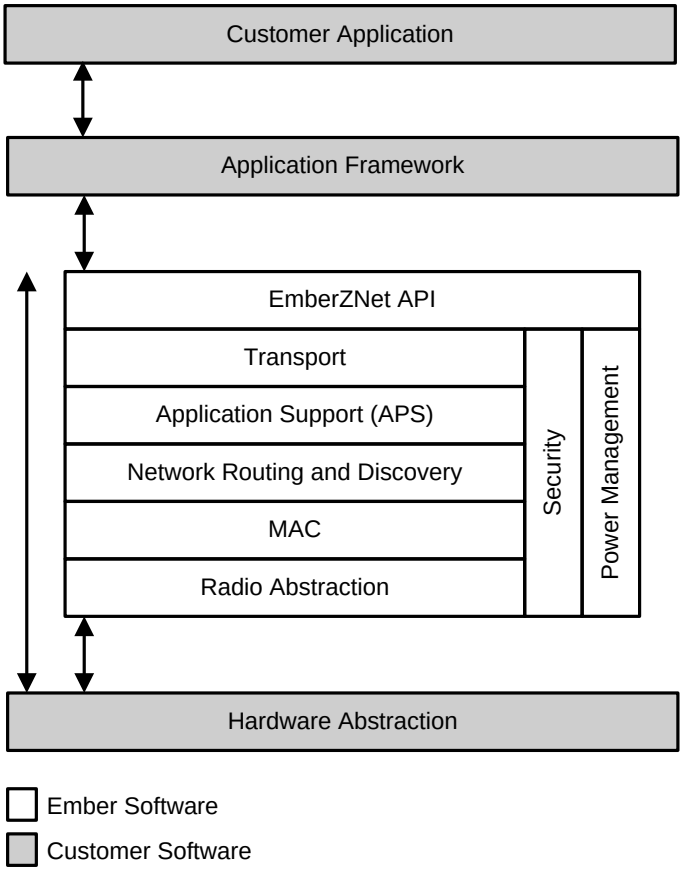
The EmberZNet 4.x is Ember's implementation of the ZigBee 2007 stack supporting the ZigBee PRO Feature set. ZigBee PRO supports mesh networks because of their increased flexibility and reliability. Consequently, all EmberZNet PRO applications must be linked with the stack library. Figure 6 1 illustrates how customer and EmberZNet PRO software interact.

In addition to this manual, additional resources are available for learning more about EmberZNet. These include:

- Ember document 120-3029-000, *Ember Application Development Fundamentals*
- Ember document 120-3030-000, *Testing and Debugging Ember Applications*

- Ember document 120-3031-000, *Advanced Application Programming with the Stack and HAL APIs*
- Platform-specific EmberZNet API References (HTML and PDF formats, provided with your installation)
- Ember sample applications (C files with many explanatory comments)
- Ember support portal (http://www.ember.com/support_index.html)

Figure 9-1. EmberZNet PRO and Customer Software Interaction



9.3 Compiler Toolchain

As mentioned above, the compiler toolchain is different based upon which Ember device and processor model you are using. The EM351/357 (EM35x) and EM250 devices, when being used as Systems-on-Chip, have customer-accessible processors. The EM35x use an ARM® Cortex™-M3 processor and an IAR toolchain, while the EM250 uses a XAP core and the xIDE toolchain. Each of these toolchains provides similar capabilities that include a compiler, linker, debugger, sample applications, and user documentation.

Note: The xIDE toolchain is licensed through Ember, while the IAR toolchain is licensed through IAR. Specific information is supplied with your software.

The EM35x and EM260, when acting as a Network Co-Processor, are designed to be a coprocessor to whatever host you might be using. One of the most commonly selected hosts is an ARM processor, though many others can be used. You can get additional information about using the EM260, as well as the EM35x and EM250, on Ember's main website (http://www.ember.com) or at the Ember Support Portal (http://www.ember.com/support_index.html).

9.4 InSight Desktop

InSight Desktop (ISD) helps you develop embedded applications through the AppBuilder application development tool and the InSight network debugging tool. The InSight network debugging tool supports over-the-air debugging and testing of your products in a live wireless environment.

With InSight, you can monitor live networks and examine debug information provided by the network devices. InSight can be used with Ember development boards to set up an easy-to-use network for developing and testing an embedded application before deploying it on prototype or production hardware. InSight can also be used with custom hardware.

AppBuilder is a rapid development tool that allows you to choose your application profile, such as Smart Energy or Home Automation, device type, cluster, and attribute set. Once you have defined your application through AppBuilder, you can generate most of your application automatically through the use of Ember-supplied application code. This code has been through extensive testing, including ZigBee certification testing so it should save you significant development, test and certification time and expense.

For more detailed information about AppBuilder and the Application Framework, see Ember document 120-3028-000, *Application Framework V2 Developer Guide.*

9.5 Peripheral Drivers

Embedded source C code is provided for drivers of peripherals such as the serial controller and analog-to-digital converter (ADC). These drivers let you incorporate standard functionality into custom applications. For more information on these drivers, see the EmberZNet API reference for your platform.

9.6 Bootloaders

The bootloader is a program stored in reserved flash memory that allows a node to update its image on demand, either by serial communication or over the air. Production-level programming is typically done during the product manufacturing process, yet it is desirable to be able to reprogram the system after production is complete. More importantly, it is valuable to be able to update the device's firmware with new features and bug fixes after deployment. The bootloading capability makes that possible. See Chapter 7 for more details.

9.7 Node Test

The nodetest applications provide low-level control of the radio and can be used to perform these tasks:

- Characterize radio performance.
- Set manufacturing and stack parameters (tokens).
- Verify proper functionality after manufacturing.
- Control the radio properly for the certification process required by many countries.

For more information about nodetest, see the relevant application note, listed in Table 9-2.

Table 9-2. NodetestApplication Notes

Document	Device
Application Note 5031, Bringing Up Custom Devices (120-5031-000)	EM250
Application Note 5041, Bringing Up Custom Devices (120-5041-000)	EM260
Application Note 5064, Bringing Up Custom Devices (120-5064-000)	EM35x

Most customers have standard product manufacturing test flows, but some do not incorporate RF testing. To address this issue, please see Ember document 120-5016-000: *Manufacturing Test Guidelines*. This document describes the different options available for integrating RF testing and characterization into your standard test flows. This application note is intended for test engineers who are moving from the early prototype development stage to the manufacturing production environment and need assistance with manufacturing test process development. This application note applies to EM250, EM35x, and EM260 devices and stack releases EmberZNet 3.0 and later.

9.8 Utilities

9.8.1 Token Utility (for EM250)

The token utility application, available with source code in the top-level /hal directory, provides read and write access to non-volatile data (tokens) that are used by the EmberZNet PRO stack and application. You can use the utility to perform these tasks:

- View the memory map of the chip's non-volatile data storage based on the CONFIGURATION_HEADER and APPLICATION_TOKEN_HEADER used at compile time.
- View and set manufacturing data and stack parameters (tokens).
- View and set custom, non-volatile data used by the application (application tokens).
- Initialize the non-volatile data area for the chip.

For more about the token utility, see its description in /app/sampleApps.htm.

Note: EM35x does not offer a version of the Token Utility application or source. The nodetest application can be used for generic viewing of manufacturing and stack tokens as well as setting stack tokens. The nodetest application is not compatible with custom application tokens.

9.8.2 Hex File Utilities

A set of tools for manipulating hex files is also available. All of these utilities (em2xx_load, em2xx_read, em2xx_convert, em2xx_patch, em3xx_load, em3xx_convert, and em3xx_buildimage) are command line (DOS console) applications. The tools are listed in Table 9-3.

Table 9-3. Hex File Tools

Tool	Description
`em2xx_load/ em3xx_load`	These utilities can be used to program the flash memory space of the EM250 through the SIF interface or the EM35x through the SWJ interface.
`em2xx_read or em3xx_load`	These utilities can be used to examine (dump) or generate a .hex file from the flash of the EM250 through SIF or the EM35x through SWJ.
`em2xx_convert`	This utility is intended for use in converting xIDE .xpv/.xdv application files into Ember's .ebl bootload format or the Intel Hex format (.hex). In addition to the representation of the application, you can include a representation of the application bootloader of the customer manufacturing tokens.
`em3xx_convert`	This utility is intended for use in converting IAR .s37 application files into Ember's .ebl bootload format. In addition to the representation of the application, you can include a representation of the application bootloader of the customer manufacturing tokens.
`em3xx_buildimage`	This utility is intended for use in manipulating EM35x file images, which includes generating Intel Hex format (.hex) files from a variety of sources, such as .s37 and .ebl files. In addition to the representation of the application, you can include a representation of the application bootloader of the customer manufacturing tokens.
`em2xx_patch`	This utility is used to program selected portions of the flash memory space of the EM2xx chips through the SIF interface. This utility can also be used to update specific portions of a .hex file
`em3xx_load`	This utility is used to program selected portions of the flash memory space of the EM35x chips through the SWJ interface.

Please refer to the Utilities Guide for the Ember chip you are using, either Ember document 120-4020-000 for the EM2xx or Ember document 120-4032-000 for the EM3xx, for detailed information.

9.9 Development Programming Tools

The InSight USB Link programmer is another useful tool (see Figure 6 7). This is a device programmer than can handle the programming requirements of the EM250 and EM260 devices in a development or small scale production environment. This programmer is USB 1.1 compatible and is fully compatible with the Ember EM250 and EM260 RCM modules and breakout boards. For more information about the InSight USB link, please contact Ember support.

Figure 9-2. InSight USB Link

The Ember InSight Adapter (ISA3) provides the programming, debugging, and data emulation capability for an EM35x-based application. Ember's EM35x chip family integrates the ARM® Cortex™-M3, 32-bit microcontroller core. The ISA3 converts between the Ember JTAG and Serial Wire (SW) commands, Packet Trace Interface, TCP/IP, and UDP for an easy-to-deploy system over 10/100 Ethernet.

As part of the EM35x Development Kit, the ISA3 connects to the EM35X module through two interfaces: the 10-pin InSight Port (ISP) and the 12-pin Insight Data Emulation Interface (DEI). These two interfaces provide access to most EM35x GPIO as well as the EM35x programming and debug I/O. See Ember document 120-2010-000, *InSight Adapter (ISA3) Technical Specification*, for more information. See also Ember documents 120-5050-000 and 120-5073-000, *Programming Options* for the EM2xx and EM35x, respectively.

Appendix A Abbreviations and Acronyms

Acronym/Abbreviation	Meaning
ACK	Acknowledgement
ADC	Analog to Digital Converter
AES	Advanced Encryption Standard
AF	Application Framework
API	Application Programmer Interface
APS	Application Support
CBA	Commercial Building Automation
CBKE	Certificate-based Key Establishment
CCM	Counter with CBC-MAC Mode for AES encryption
CCM*	Improved Counter with CBC-MAC Mode for AES encryption
CLI	Command Line Interface
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
CW	Continuous Wave
EBL	Ember Bootload
EEPROM	Electrically Erasable Programmable Read Only Memory
EHF	Extremely High Frequency
ESP	Ember Serial Protocol
EZSP	Ember ZNet Serial Protocol
GPIO	General Purpose I/O (pins)
GUI	Graphical User Interface
HA	Home Automation
HAL	Hardware Abstraction Layer
HC	Health Care

Acronym/Abbreviation	Meaning
HF	High Frequency
HVAC	Heating, Ventilation, and Air Conditioning
HTML	Hypertext Markup Language
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISD	InSight Desktop
JTAG	Joint Test Action Group
LAN	Local Area Network
LF	Low Frequency
LQI	Link Quality Indicator
LSB	Least significant bit
MAC	Medium Access Control
MF	Medium Frequency
MIC	Message Integrity Code
MSB	Most significant bit
MSP	Manufacturer-specific Profile
MTOR	Many-to-one Routing
NACK	Negative Acknowledge
NCP	Network Coprocessor
NIST	National Institute of Standards and Technology
NWK	Network (layer)
OEM	Original Equipment Manufacturer
OQPSK	Offset Quadrature Phase-shift Keying
OTA	Over-the-Air
PA	Power Amplifier
PHY	Physical Layer
POSIX	Portable Operating Standard (for Unix)
PSK	Phase-Shift Keying
PTI	Packet Trace Interface
RAM	Random Access Memory

Acronym/Abbreviation	Meaning
RCM	(Reader control module?)
RF	Radio Frequency
RNAP	Remote Node Access Protocol
Rx	Receive
SE	Smart Energy
SHF	Super High Frequency
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SWJ	Serial Wire and JTAG Interface
TA	Telecom Application
TCP	Transmission Control Protocol
Tx	Transmit
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol
UHF	Ultra High Frequencies
VCO	Voltage Controlled Oscillator
VHF	Very High Frequency
VLF	Very Low Frequency
WPAN	Wireless Personal Area Network
WSN	Wireless Sensor Network
XML	Extensible Markup Language
ZC	ZigBee Coordinator
ZCL	ZigBee Cluster Library
ZED	ZigBee End Device
ZDO	ZigBee Device Object
ZR	ZigBee Router