UM0923
User manual

EmberZNet™ application developer guide

1 Introduction

Purpose

This document:

- Describes how to design and implement a project using the EmberZNet ZigBee PRO compliant EmberZNet network software.
- Contains introductory information on radio propagation and embedded mesh networking topologies, and details on networking that are important for understanding how the system functions and making the correct design decisions.

STMicrolectronics recommends that you read this document from beginning to end, because later chapters rely on information in previous chapters. You can then refer to specific chapters as necessary during your development process.

Audience

This document is intended for project managers and for software engineers who are responsible for building a successful embedded mesh networking solution using the STMicrolectronics radios, the EmberZNet network stack, and tools.

This document assumes that the reader has a solid understanding of embedded systems design and programming in the C language. Experience with networking and radio frequency systems is useful but not expected.

Getting help

STM32W108 kit customers are eligible for training and technical support. You can use the STMicrolectronics website, www.st.com/mcu, STM32W section, to obtain information about all STMicrolectronics products and services, and to sign up for product support.

Documentation conventions

Table 1. Documentation conventions

Notation	Meaning	Example
Bold	A GUI label.	**Node Key**
UPPERCASE	A keyboard key	ENTER
|	Delimits a hierarchy of menu options, which ends with the option to choose.	Open | Save
`Courier`	Identifies file names and program identifiers, such as constants and function names.	`EMBER_SLEEPY_END_DEVICE` `serial.h, emberStartScan()`

Contents

2 Wireless sensor networks overview

2.1 Overview

As embedded system design has evolved, the need for networking has become a basic design feature. 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 the 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, HVAC control, fire/smoke/CO alarms, remote doorbells, humidity monitors, energy usage monitors, and countless others 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 to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations.

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 (WPAN) 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 low power and are thus 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 is most commonly use to refer 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 LAN, or Local Area Network. 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.

Wireless networks have significantly impacted the world as far back as World War II. With the use of wireless networks, information could be sent oversees or behind enemy lines easily and quickly and was more reliable. Since then wireless networks have continued to develop and their uses have significantly grown. Cellular phones are part of huge wireless network systems. People use these phones daily to communicate with one another. Emergency services such as the police department utilize wireless networks to communicate important information quickly. People and businesses use wireless networks to send and share data quickly whether it be in a small office building or across the world.

In the control world, embedded systems have become commonplace for operating equipment using local special purpose computer hardware. Wired networks of such devices have become commonplace 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. Wireless networking of embedded systems (that is Embedded Networking) have become commonplace. However, the costly embedded networking solutions have only been justifiable in high-end applications where the costs are a secondary consideration. For low cost applications with low data rate communications requirements, there has not been a good standardized solution until the IEEE 802.15.4 standard for wireless personal area physical layer and MAC was released in 2003. The ZigBee group was formed to establish networking and application level standards on top of the IEEE 802.15.4 standards to allow flexibility, reliability and interoperability.

Although wireless networks allow you to eliminate messy cables and enhance installation mobility, there is a downside from 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 channel sharing mechanisms at the medium access control layer (MAC) to allow sharing of channels. In addition, the purpose of the mesh networking within ZigBee is to provide redundant paths for data within the network that are automatically rediscovered and used to avoid interference in a local area.

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. the application designer needs to ensure their system data rates are within what is achievable with the system being used.

Wireless network security is a unique problem since the data can easily be overheard by eavesdropping devices. ZigBee has a set of security services designed around AES 128 encryption to ensure the system designer has a choice of security levels based on the needs of the application. Careful design around these standards helps maintain high levels of network security.

Other networking standards exist such as Bluetooth. But 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 simple 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 and other large distributed networks.

2.3 ZigBee

Section 3: ZigBee overview provides an in-depth discussion of the ZigBee industry group and its efforts to standardize IEEE 802.15.4 based applications. The current version as of this writing is the IEEE 802.15.4-2006 standard.

2.4 Radio fundamentals

Radio has been a part of our world since Marconi and DeForrest's work at the opening of the 20th Century. But what exactly is radio and what does it have to do with networking?

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 sounds 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 that the radio spectrum in the United States is 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:

Table 2. radio sub-spectrums

Sub-spectrum	Description
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.

2.4.1 Frequency bands

As mentioned, the radio spectrum is regulated by government agencies and by international treaties. Most transmitting stations require a license to operate, including commercial broadcasters, military, scientific, industrial and amateur radio stations. 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. There are three frequency bands that 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 an 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 5200Mhz band has three sub-bands, the lowest being for indoor home use only, while the 5800Mhz 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.

Note: A detailed discussion of ZigBee and the ZigBee Standard is the subject of Section 3.

2.4.2 Signal modulation

So, we can send an electrical signal out in the air, but we must make the electrical signal behave in a way that allows it to transfer intelligent information. This process is called modulation, but you can think of it also as a way to encode information to be transmitted to a receiver that will decode, or demodulate, 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 is 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 STM32W108 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. This requires the receiver to be able to compare the phase of the received signal to a reference signal - such a system is termed coherent.

Note: Fortunately, designing with the STM32W108 does not require you to be an expert with this technology; just being aware of it is enough.

2.4.3 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. You can not take an antenna from a 2400 MHz radio and use it effectively on a 5800 MHz radio, or vice versa.

There are two fundamental types of antennas, which, with reference to a specific three dimensional (usually horizontal or vertical) plane:

- Omni-directional (radiates equally in all directions)
- Uni-directional (or, 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 omnidirectional antenna is desired to provide as many communication paths as possible.

2.4.4 How far signals travel

We can tell how far a radio signal will travel, and get an idea of how much information we can transmit 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 type of physical/electrical obstructions are in the way.

2.4.5 Radio transmit power - It's all about the dB

Radio transmit power is measured in decibels, or “dB.” Sometimes you may hear radio transmit power talked about in terms of watts. Instead of using watts you can convert wattage to dB. Doing so lets you calculate radio links using simple addition and subtraction.

For example, a typical power amplified Wireless radio card transmits at 100milli-watts (or mW). Instead of 100 mW, we say it has 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 1mW (2mW to be specific). The standard output power of the STM32W108 device is +8 dBm (+ 0.5 dBm in Boost mode). Using a power amplifier module can increase the transmit power to 20 dBm, but this requires more power to operate.

2.4.6 Amount of radio signal needed to “hear”

The radio also needs to be able to hear a radio signal at a certain level. As the radio signal travels through the air, it weakens (much like shouting at someone from a mile away). The minimum signal strength required for a receiver to understand the data is called the receive sensitivity.

When a radio signal leaves the transmitting antenna your dB will be a high number (for example: 20 dB). As it travels through the air, it loses strength and will drop to a negative number. This is why the amount of power a receiver needs is often rated as low as -90 dB.

The use of negative numbers can be confusing at first. Just remember, 20 is higher than 0, and -20 is lower than 0. Thus, if you can achieve a signal level of -75 dB and your radio needs -95, you have 20 dB of extra signal to accommodate interference and other issues.

2.4.7 How far can the radio signal go?

Now that we know how much power we can put out, and how much we need, we can figure out how much radio signal will be available at the receiving end. The way we figure this out is to determine how much loss is present between the radio transmitter and receiver.

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

Table 3. Free space loss

What	Add or subtract it	The value (all in dB!)
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

Based on these calculations, we can guesstimate that 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 will yield 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.

So now that you've tried the manual calculation, you'll be happy to know there are easier ways to figure this out. The STM32W108 Kits come with a nodetest application that can be used to perform empirical range testing for your embedded wireless network in virtually any environment. It is recommended that basic range testing be conducted in the expected environment to evaluate whether extended range is required.

*Note: 1 dBm = 10*log10(P/ 0.001)*

2 A reference table of common Free Space Losses is available on the BC Wireless Website at http://www.bcwireless.net/moin.cgi/FreeSpaceLossTable. A spread sheet for Microsoft Excel and Open Office is also available from BC Wireless.

2.5 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 topologies. The term Internet (note the capitalization) is often used to refer to the largest worldwide system of networks, also called the Worldwide Web. The basic protocol used to implement the WorldWide 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 AppleTalk Data Stream Protocol (ADSP) uses the Datagram Delivery Protocol (DDP) to encapsulate the data and deliver it over an AppleTalk network. The protocol that uses the services of an underlying protocol is said to be a client of the lower protocol; for example, ADSP is a client of DDP. A set of protocols related in this fashion is called a protocol stack.

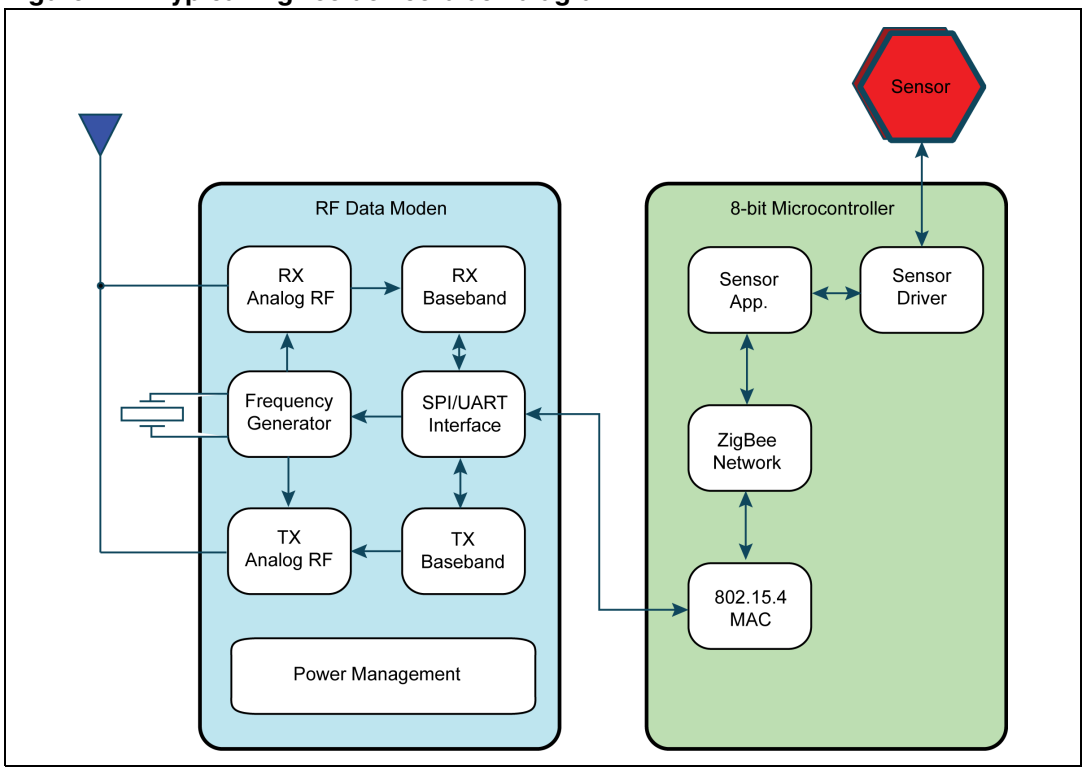
2.6 Wireless networking

Wireless networking mimics the wired network, but replaces the data interconnection medium from wire to a radio signal (that is, wireless). 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 foundations to enable reliable networking within the scope of their unique environments.

2.7 EmberZNet ZigBee devices

The STM32W108xx IC family and software (the EmberZNet Stack and development tools) facilitate implementation of a Wireless Personal Area Network (WPAN) of devices for sensing and control applications. The diagram in *Figure 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.

Figure 1. Typical ZigBee device block diagram



The STM32W108 provides both the RF and microcontroller portions of *Figure 1*. EmberZNet networks support the device types listed in *Table 4*.

Table 4. EmberZNet device types

Device type	Description
`EMBER_COORDINATOR` (ZigBee coordinator)	Relays messages and can act as a parent to other nodes. Every personal area network (PAN) must be started by a node acting as the coordinator. 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.

Table 4. EmberZNet device types

Device type	Description
`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)	A sleepy end device that can move through the network.

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. 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.

2.7.1 Network formation and operation

The coordinator initiates network formation. In a mesh network, after forming the network, the coordinator can function as a router. The EmberZNet 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.

A device finds a network by scanning channels. When a device finds a network with the correct stack 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.

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. There are two general approaches, 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.

The EmberZNet stack uses on-demand route discovery.

In a ZigBee tree 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 at the same depth of the network tree-that is to say, they are the same number of hops away from the coordinator node. If they join the network through different parent nodes, they can only route messages to one another 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.

In an EmberZNet 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 overview

3.1 Introduction

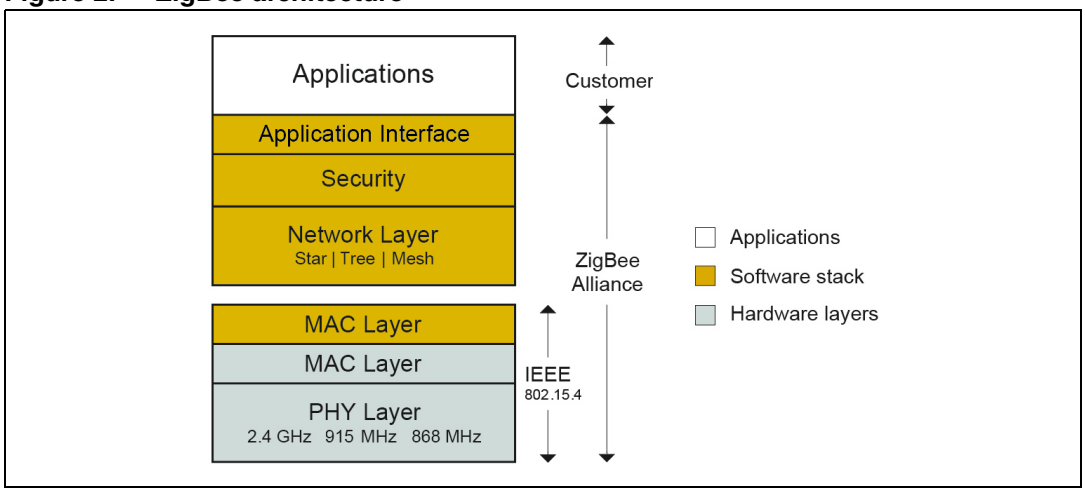
ZigBee is an alliance of companies working together to enable reliable, cost effective, low power, wireless networked, monitoring and control products based on an open global standard.

The ZigBee Alliance is operated by a set of promoter companies that make up the Board of Directors. These currently include ST Microelectronics, Ember, Philips, Schneider Electric, Siemens, Cellnet, Itron, Tendril, Eaton, Mitsubishi Electric, Samsung, Honeywell, Texas Instruments, Motorola, Huawei 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. Business is conducted in an open manner resulting in standards 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.

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

Because ZigBee is committed to open and interoperable devices, standards have been developed from the physical layer through the application layer as shown in *Figure 2*. At the physical and MAC 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 STMicroelectronics is committed to supporting these as they are available.

Figure 2. ZigBee architecture



EmberZNet stack provides a standard networking API based on the ZigBee specification across the STM32W108 platforms. The EmberZNet stack also provides sample applications using ZigBee Application profiles to allow rapid development of customer applications.

3.1.1 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, health care, automated meter reading and personal health care monitoring. The general characteristics for a ZigBee network are as follows:

- Low power - Devices can typically operated for several years on AA type batteries using suitable duty cycles.
- 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.
- 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 and 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 and the simple protocols for forming and joining networks allow systems to self configure and fix routing problems as they occur.

3.1.2 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 40/20 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 adopted the 802.15.4 - 2003 version of the standard. The IEEE has since issued a 2006 version of this standard that has not yet been adopted by ZigBee.

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 specific changes to the 802.15.4 MAC that are documented in Annex D of the ZigBee specification.

The 802.15.4 medium access control (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 retries and acknowledgement of messages for reliable communications between adjacent devices. These underlying mechanisms are built upon by the ZigBee network layer to provide reliable end to end communications in the network. The 802.15.4 standard is available from www.ieee.org

3.1.3 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 there are some third party software stacks that have been developed. Typically

reference designs for the hardware and sample applications for the software are provided by the hardware and software provider. Based on these, a hardware developer can customize the hardware to their specific needs. Alternatively, there are a number of module providers that can deliver compact and low cost modules that can use used.

Because of the embedded nature of typical ZigBee applications, software application development is typically interrelated with the hardware design to provide an optimal solution.

The software developer can develop a complete ZigBee application extending the sample application provided by the stack provider Alternatively, there are a number of third party software development firms that specialize in development of ZigBee applications and can assist in new product development.

The EmberZNet stack includes a number of typical sample applications as well as some common utilities or tools. These include:

- Reference application for the Home Automation and Smart Energy profiles using the ZigBee Cluster Library
- Over the air bootloaders to allow upgrading of 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.

3.1.4 ZigBee network topologies

ZigBee supports a variety of network topologies. 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.

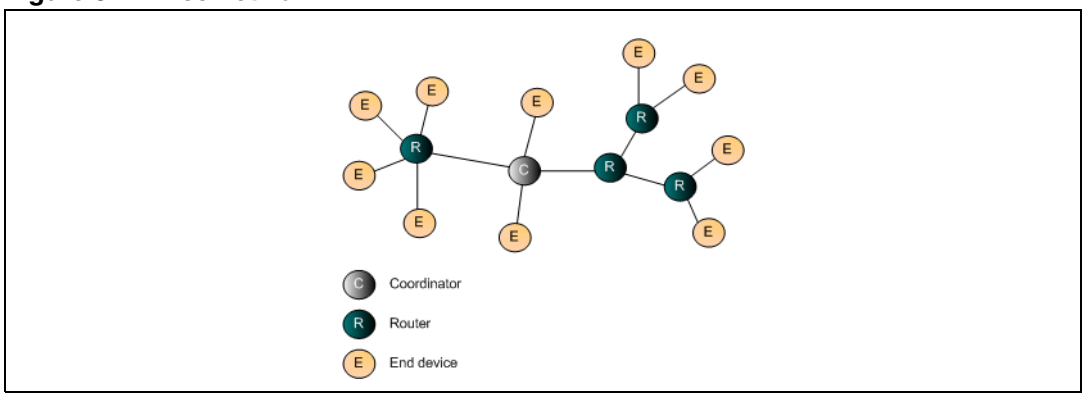
ZigBee supports two basic topology types:

- Tree network
- Mesh network

Tree network

The EmberZNet tree topology (see *Figure 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 and instead always uses the mesh topology noted below.

Figure 3. Tree network



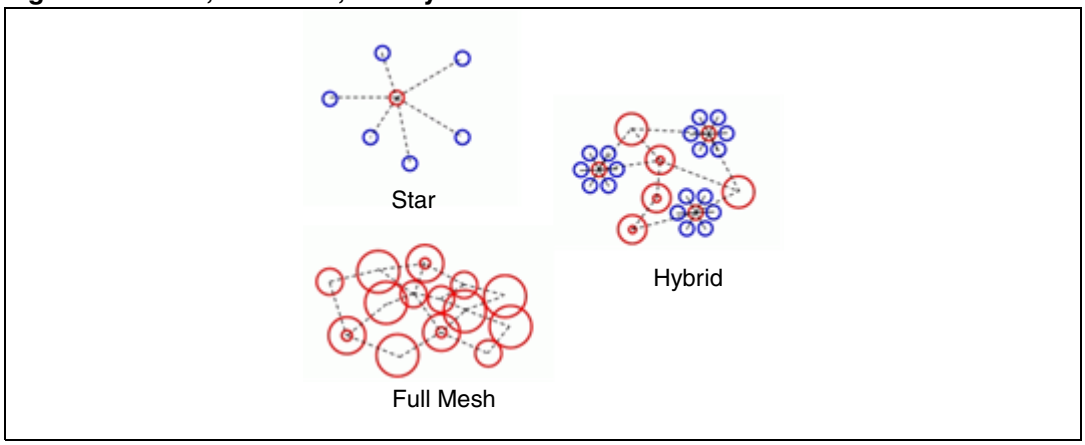
Mesh network

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

EmberZNet supports three types of mesh network topologies, as shown in *Figure 4*:

- Star network
- Full Mesh network
- Hybrid mesh network

Figure 4. Star, full mesh, and hybrid mesh networks



Note: Blue devices in Figure 4 are end devices and can be sleepy or mobile.

Star network

In a Star network, one hub is the central point of all communications. The hub can become bottle-necked 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.

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.

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.

3.1.5 Network node types

The ZigBee specification supports networks with one coordinator, multiple routers, and multiple end devices within a single network:

Coordinator

A ZigBee coordinator (ZC) is responsible for forming the network. This included selection of an appropriate channel after scanning available channels and selecting an extended PAN ID. 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.

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

For some applications, the coordinator may have added responsibilities such as being the trust center or network manager. These choices are up to the application developer and in some cases these choices 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.

Routers

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.

End devices

End devices (ZED) are leaf nodes: they communicate only with 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. 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 EmberZNet modification to the basic ZigBee sleepy end device to provide added capabilities and management of mobile devices.

The EmberZNet stacks support sleepy and non-sleepy end devices.

3.1.6 ZigBee routing concepts

ZigBee has several routing mechanisms that can be used based on the network and expected traffic patterns. The application designer has choices of which mechanism to use and this selection should be made as part of the system architecture and design. In actual practice one application may use several of these routing mechanisms because some devices are performing one-to-one communications while all devices are also

communicating to a central monitoring device. The types of routing discussed below are tree routing, table routing, multicast routing, broadcasts and many-to-one/source routing.

In the ZigBee 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.

Table (or mesh) routing also exists in the ZigBee stack. This table routing is a derivative of AODV next hop routing. A node can send a unicast message table routing by requesting route discovery-either as needed or by force-in the APS unicast options for that message. When a node sends a message to a destination for which a table route has been discovered, the tree stack first attempts tree routing delivery; if tree routing delivery fails and the message specifies the APS Retry option, the stack falls back on 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 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. Each node uses its own routing table to determine the next hop that is required to deliver messages to any other node. If a route fails, a route error is sent back to the originator of the message who can then rediscover the route.

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 and therefore should be used only as necessary in applications because over use of broadcast mechanisms can degrade network performance. A multicast message is never acknowledged.

Broadcast routing is a mechanism to send a message to all devices in a network. There are network level broadcast options to send to routers only or also to send broadcasts to sleeping end devices. A broadcast message is repeated by all powered devices in the network 3 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.

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. Given the memory limited devices often used in ZigBee networks, these large tables are undesirable. Under many-to-one routing, the central device sends a single route discovery that established a single route table entry in all routers to provide the next hop to the central device. All devices in the network then have a next hop path to the central device 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 similar increase in route table size. Instead, incoming messages to the central device first use a route record message to store the next hops used. The central device then stores these next hop routes in a route record table. Outgoing messages include this route record 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 central device does require some additional memory if it is storing these route records.

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

3.1.7 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.

It is not recommended 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.

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) - defining devices for typical residential and small commercial installations
- Commercial Building Automation (CBA) - defining devices for large commercial buildings and networks.
- Smart Energy (SE) - For utility meter reading and interaction with household devices
- Telecom Application (TA) - Wireless applications within the telecom area.
- Wireless Sensor Network Applications (WSN) - Wireless sensor networks
- Personal Home Health Care (PHHC) - Monitoring of personal health in the home environment

A ZigBee Cluster Library (ZCL) forms a generic basis for some of these application profiles. This library exists to allow 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. There are two types of application profiles administered by the Alliance:

- Public Application Profiles are developed by members of the ZigBee Alliance to assure 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 ID's 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.

ZigBee addressing schemes

ZigBee contains two network addressing schemes built into the stack. It also has the ability to assign addresses manually from a commissioning tool.

The ZigBee 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 this number of 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 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 to 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, there is also intended to be provisions for a commissioning tool for setup and configuration of networks. These commissioning tools can be used to provide addresses manually.

Under the EmberZNet implementation of ZigBee, we recommend using ZigBee Pro stack profile with the alternate address assignment mechanism. We have more than 3 years of field experience with this system and consider it very stable and robust.

ZigBee messaging options

Please see *Section 3.1.6: ZigBee routing concepts*.

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 5. 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 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 manufactures specific application. Those application 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. STMicroelectronics sends all chips and stack releases through compliance testing.

All of the ZigBee test plans are developed and tested using at least 3 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).

Applying ZigBee

There are a set of design decisions to be made in any ZigBee implementation. While these areas of application development are covered in more detail in later chapters, the initial design choices are as follows:

1. Implement ZigBee at the board level or as a module - There are a number of ZigBee modules 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. Selection of a ZigBee Platform - The available ZigBee platforms include the STM32W108. The STM32W108 is a single chip implementations designed for low cost implementations.
3. Select a Bootloader - There are several choices available for in the field upgrading of devices. The simplest choice is to not upgrade devices once deployed (no bootloader). However, this is not recommended as there are often issues that are found during initial field trials or beta testing that require software updates. There are two bootloaders provided by EmberZNet and commonly used. One is a stand alone bootloader that provides for using and upgrading the entire flash contents. This bootloader is limited to one hop transfer of data. The application bootloader can be used across multiple hops under the application control but requires extra flash to store the image being bootloaded.
4. Designing 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. Developing and Debugging 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.
6. Field Trials - Once lab and office testing is completed, it is recommended that more extensive field trials be conducted at typical expected installations. These installations should be carefully monitored and evaluated for improvements to the application.
7. Manufacturing Test - Development of production level hardware requires consideration of the manufacturing testing to be conducted and how the application supports these tests. EmberZNet has an embedded manufacturing library to assist in this testing.

4 Designing an Application

4.1 ABCs of application design

It is an unavoidable reality that you cannot simply begin coding your application until you have completed some vital preliminary tasks. These include your Network Design and your System Design. However, you cannot design your system or network until you understand the scope of requirements and capabilities in a basic node application. This chapter will begin by describing the task-based features that your application must include.

Design is an iterative process. Each iteration has a ripple effect throughout the system and network designs. In this iterative process, the Network Design is the first step. Different network topologies have their own strengths and weaknesses. Some topologies may be totally unsuitable for your product, while some may be marginally acceptable for one reason or another. Only by understanding the best network topology for your product can you proceed to the next step: System Design.

Your System Design must include all the functional requirements needed by your final product. These must include hardware-based requirements, network functionality, security, and other issues. The question of whether or not you will seek ZigBee Certification for your product is another important factor in your system design because it will impose specific functional and design requirements on your application code. Only once you have a fully defined set of requirements and a system design that implements these requirements can you proceed to the next step in the process: Application Coding.

Golden Rules

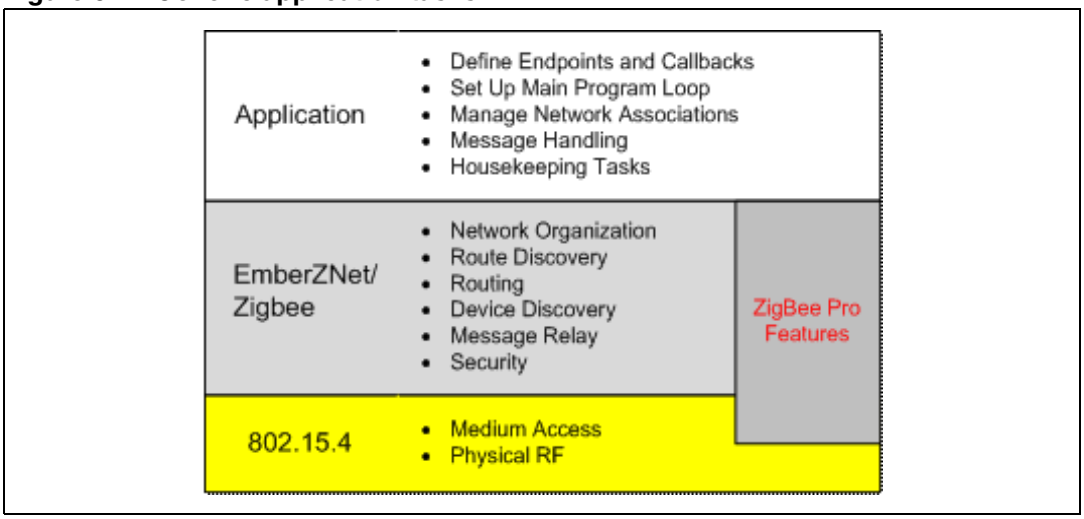
- Your network will probably be either a sensor-type or a control-type network, or maybe have elements of both.
- The potential network bandwidth and latency are topology-dependent.
- The best solution to certain challenges encountered by everyone are likely to be application-specific.

Your Application Software will implement your system design. It will also use the EmberZNet stack software to provide the basic functionality needed to create your ZigBee Wireless Personal Area Network (WPAN) product. Testing will follow completion of your application software to confirm that it functions as intended. And, as in most design environments, you will repeat the design/test cycle many times before completing your product design.

4.2 Basic application design requirements

The typical EmberZNet embedded networking application must accomplish certain generic tasks. *Figure 6* summarizes these tasks.

Figure 6. Generic application tasks



4.2.1 Scratch-built or 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. This approach is recommended and tools and examples are provided for this purpose.

EmberZNet supplies several sample applications that are installed along with your EmberZNet stack software. Some of these samples are intended to illustrate specific features or functional applications of EmberZNet.

4.3 Basic application task requirements (scratch-built)

If you choose to develop your own application from scratch, the following sections provide general design guidelines for you to follow. *Figure 6* describes five major tasks that the generic networking application must perform in a ZigBee environment. In this section we will examine each of these tasks in more detail.

4.3.1 Define endpoints, callbacks, and global variables

Your main source file for your application must begin with defining some very important parameters. These parameters involve endpoints, callbacks, and some specific global variables.

Endpoints are required to send and receive messages, so any device (except a basic network relay device) will need at least one of these. As far as how many and what kinds, that's entirely up to the user. *Table 5* lists endpoints that must be defined in your application. Look at the article on for a further discussion of endpoints, endpointDescriptions, cluster IDs, profiles, and related concepts.

Table 5. Required endpoint stack global variables

Endpoint	Description
`int8u emberEndpointCount`	Variable that defines how many user endpoints we have on this node.
`EmberEndpointDescription PGM endpointDescription`	Typical endpoint descriptor; note that the PGM keyword is just used to inform the compiler that these descriptions live in flash because they don't change over the life of the device and aren't accessed much.
`EmberEndpoint emberEndpoints[ ]`	A global stack array that defines how these endpoints are enumerated; each entry in the array is made up of the "identifier" field of each EmberEndpoint struct.

Required callback handlers are listed in *Table 6*. Full descriptions of each function can be found in the EmberZNet Stack API Guide.

Table 6. Required Callback Handlers

Callback Handler	Description
`emberMessageSentHandler( )`	Called when a message has completed transmission. A status argument indicates whether the transmission was successful or not.
`bootloadUtilQueryResponseHandler( )`	Only required if linking in the bootloader library. When a device sends out a bootloader query, the bootloader query response messages are parsed by the bootloader-util library and handed to this function. The application reacts as appropriate and sends a query response.
`emberIncomingMessageHandler( )`	Called whenever an incoming message occurs.
`emberStackStatusHandler( )`	This is called whenever the stack status changes. A switch structure is usually used to initiate the appropriate response to the new stack status.
`emberScanCompleteHandler( )`	This function is called when a network scan has been completed.
`emberNetworkFoundHandler( )`	This function is called when a network is found during a network scan.

Of clusters and endpoints and profiles...

So, how do endpoints, clusters, profiles, endpoint descriptions and similar things relate to one another?

In object-oriented terms, each endpoint is an instance of an endpoint descriptor, which is like the “class” describing the attributes/functionality of an endpoint. That descriptor “inherits” its capabilities from the application profile (be it a private, proprietary or official/public ZigBee one) and the device type is specified in the descriptor.

Let's consider an example of a private Application Profile (or “App Profile”) called “Animals.” The device types in this app profile would be sub-classes of the whole, like “Zebra”, “Giraffe”, and “Hippo”, and each of those has some set of behaviors (like, say, “eating”, “sleeping”, “walking”) and attributes (maybe “weight”, “height”, “number of teeth”, “age”, and “neck size”). These behaviors and attributes are analogous to the cluster IDs that ZigBee's profiles define to enumerate the various behaviors/attributes of the devices in the profile.

Many of these behaviors/attributes may overlap across different device types. Their values may not be the same, but the presence or lack of these behaviors/attributes may be common across multiple devices and even across different profiles (like how “People” can have “weight” and “height” attributes as well as “eating” and “sleeping” capabilities). The ZigBee Cluster Library (ZCL) is an attempt to categorize these attributes/behaviors in general ways; like having a “Bodily Functions” cluster to encompass things like “height”, “weight”, “eating”, “sleeping”, etc, or an “Emotions” cluster to encompass attributes/behaviors like “be aggressive”, “be nurturing”, “serotonin level”, “adrenaline level”. This allows completely different profiles to still reuse a lot of the same categories and specific attributes and behaviors; like the way that a “People” profile could still utilize many of the same concepts that an “Animals” profile would utilize.

Suppose, on my node (the equivalent of, say, a family unit, for the purposes of this analogy), I have 3 endpoints:

1. An African-American woman named Harriet.
2. A male Siamese cat named Fluffy.
3. A Caucasian boy named Brad.

So the descriptors might look something like this:

```
EmberEndpointDescription PGM harriet =
{
PEOPLE_PROFILE_ID, //some 16-bit identifier that ZigBee uses to
designate our "People" protocol
WOMAN_DEVICE_ID, //some 16-bit identifier that our People profile
uses to designate a "woman"
(AMERICAN | AFRICAN | SENIOR_CITIZEN), //some bitmask that serves as
a further specifier for what version/sub-type of device (woman) this
is (which allows others to infer some presence/ lack of
behaviors/attributes on our part)
HARRIET_INPUT_CLUSTER_COUNT, // how many different kinds of inbound
requests can Harriet understand (and can act upon) when asked?
HARRIET_OUTPUT_CLUSTER_COUNT // how many different things is Harriet
capable of reporting to / requesting of others?
};
EmberEndpointDescription PGM fluffy ={ANIMALS_PROFILE_ID,
CAT_DEVICE_ID, (MALE | SIAMESE), FLUFFY_INPUT_CLUSTER_COUNT,
FLUFFY_OUTPUT_CLUSTER_COUNT
};
EmberEndpointDescription PGM brad = // note that the "PGM" keyword
is just to tell the compiler that these things live in flash because
they don't change over the life of the device and aren't accessed
much
{PEOPLE_PROFILE_ID, MAN_DEVICE_ID, (AMERICAN | CAUCASIAN |
ADOLESCENT), BRAD_INPUT_CLUSTER_COUNT, BRAD_OUTPUT_CLUSTER_COUNT};
int16u
harrietInputClusterList[HARRIET_INPUT_
CLUSTER_COUNT] = {
0x0101, // sleep action
0x0102, // eat action
0x0201, // current age request
0x202, // current emotion request
…etc…
```

```
} ;
```

Like the ZCL, the cluster ID ranges have been made to correspond to a group (“cluster”) of related message types (for example: 0x0100-0x01FF for requests to perform behaviors, 0x0200 - 0x02FF for requests of attributes, 0x300 - 0x3FF for event notifications).

Other related stack global constructs include: emberEndpointCount describes how many of these endpoints we have in this family (on this node): 3, in this case. The emberEndpoints[] array defines how these endpoints are enumerated in the family. So, based on the order we listed them at the top of this example, the declarations would look something like this:

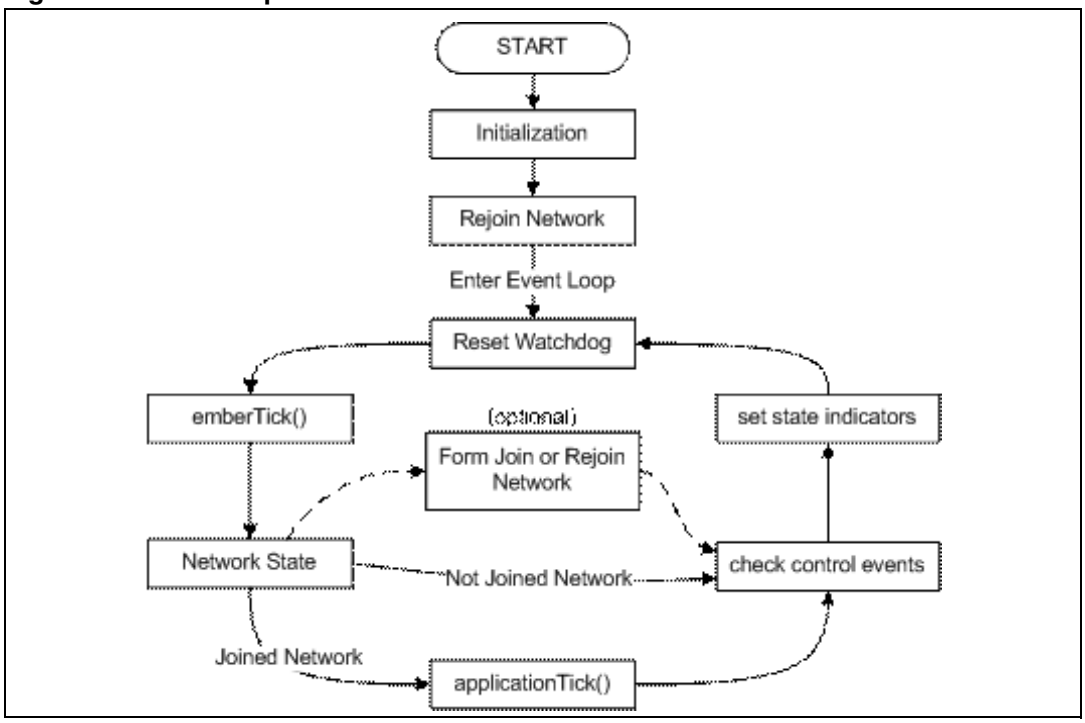
```
int8u emberEndpointCount = 3;
EmberEndpoint emberEndpoints[ ] = { // should contain 3 elements, as
per the count we promised above
{1, &harriet, harrietInputClusterList, harrietOutputClusterList},
{2, &fluffy, fluffyInputClusterList, fluffyOutputClusterList},
{3, &brad, bradInputClusterList, bradOutputClusterList
};
```

Having said all this, it's worth noting that in EmberZNet these descriptors really are just kept around for informational purposes, should someone happen to ask. What you put in these fields [how much and how accurate] is entirely up to you, the firmware engineer. The stack only cares about how many endpoints you have described by the user-defined stack global, `emberEndpointCount` and what endpoint numbers they occupy (as defined by the “identifier” field of each `EmberEndpoint` struct that populates the user-defined global stack array, `emberEndpoints[]`).

4.3.2 Setup main program loop

The main program loop is central to the execution of your application. *Figure 7* describes a typical EmberZNet application main loop.

Figure 7. Main loop state machine



Initialization

Among the initialization tasks, any serial ports (SPI, UART, debug or virtual) must be initialized. It is also important to call emberInit() before any other stack functions (except initializing the serial port so that any errors have a way to be reported). Additional tasks include the following.

Prior to calling `emberInit()`:

- Initialize the HAL
- Turn on interrupts
- Call the `halGetResetInfo()` function to check for reset information

After calling `emberInit()`:

- Try to rejoin the network if previously connected (see `emberNetworkInit()` in the Stack API Guide)
- Set the security key (see the function `setSecurityKey()` in the sample applications)
- Initialize the application state (in this case a sensor interface)
- Set any status or state indicators to initial state
- Set the bootloading condition, if used

Event loop

The example in *Figure 7* is based on the sensor.c sample application. In that application, joining the network requires a button press to initiate the process; your application may use a scheduled event instead. The network state is checked once during each circuit of the event loop. If the state indicates “joined,” then the `applicationTick()` function is executed. Otherwise, the execution flow skips over to checking for control events.

This application uses buttons for data input that are handled as a control event. State indicators are simply LEDs in this application, but could be an alphanumeric display or some other state indicator.

The `applicationTick()` function provides services in this application to check for timeouts, check for control inputs, and change any indicators (like a heartbeat LED). Note that `applicationTick()` is only executed here if the network is joined.

The function `emberTick()` is a part of EmberZNet. It is a periodic tick routine that should be called:

- In the application's main event loop
- After `emberInit()`

The STM32W108xx has a watchdog timer that should be reset once during each circuit through the event loop. If it times out, a reset will be triggered. By default, the watchdog timer is set for 2 seconds.

4.3.3 Manage network associations

The application is responsible for managing network associations. The tasks involved include:

- Detecting a network
- Joining a network
- Forming a new network

Detecting a network

Detecting a network is handled by the stack software using a process called Active Scan, but only if requested by the application. To do so, the application must use the function `emberStartScan()` with a scantype parameter of `EMBER_ACTIVE_SCAN`. This function will start a scan and return `EMBER_SUCCESS` to signal that the scan has successfully started. It may return one of several error values that are listed in the online API documentation.

Active Scanning walks through the available channels and detects the presence of any networks. The function `emberScanCompleteHandler()` is called to indicate success or failure of the scan results. Successful results are accessed through the `emberNetworkFoundHandler()` function that reports the following information:

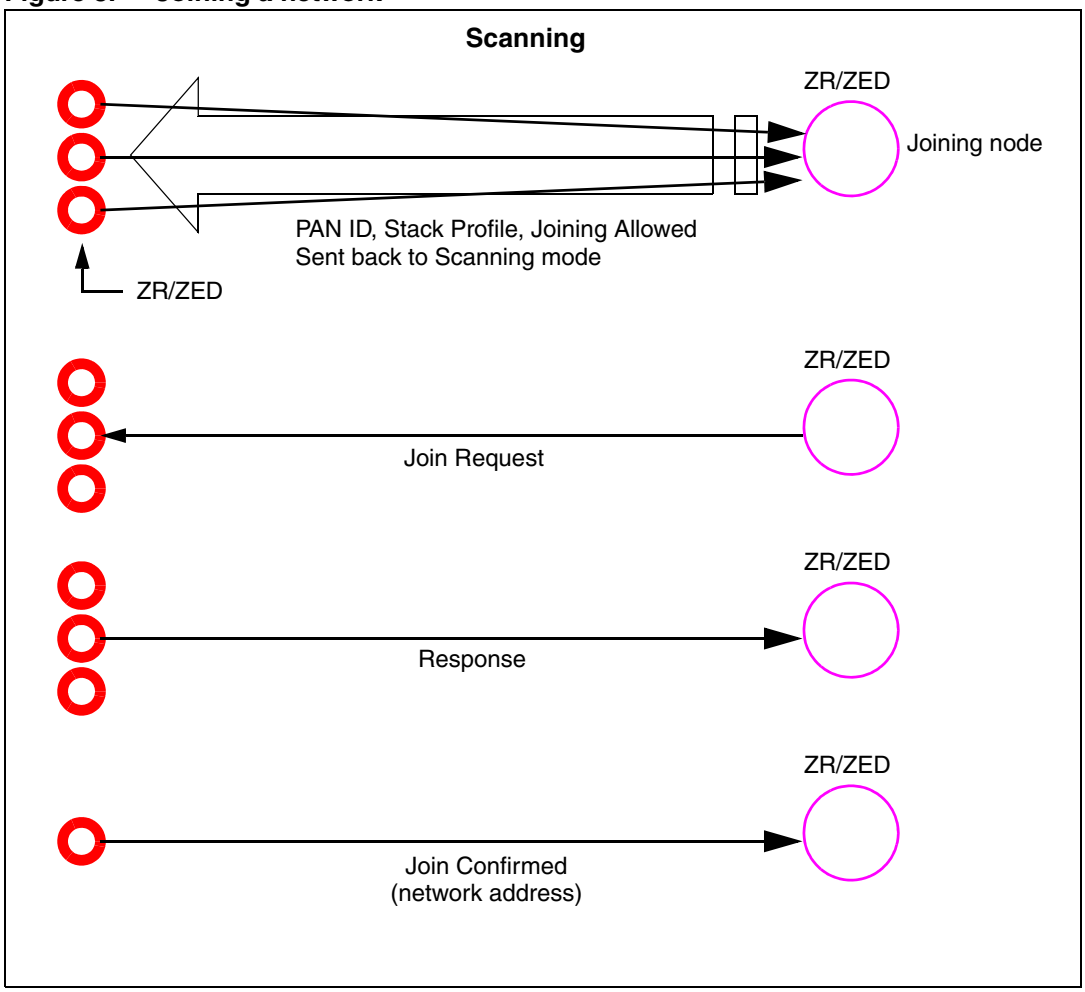
Table 7. Detecting a network

Function Parameter	Description
`channel`	The 802.15.4 channel on which the network was found.
`panId`	The PAN ID of the network.
`extendedPanId`	The Extended PAN ID of the network.
`expectingJoin`	Whether the node that generated this beacon is allowing additional children to join to its network.
`stackProfile`	The Zigbee stack profile number of the network. 0 = private, 1 = ZigBee, 2 = ZigBeePro.

Joining a Network

To join a network, a node must generally follow a process like the following, illustrated in *Figure 8*:

Figure 8. Joining a network



1. New ZR or ZED scans channels to discover all local (1-hop) ZR or ZC nodes that are join candidates.
2. Scanning device chooses a responding device and submits a join request.
3. If accepted, the new device receives a confirmation that includes the network address.

Joining a network involves calling the `emberJoinNetwork()` function. It causes the stack to associate with the network using the specified network parameters. It can take ~200ms for the stack to associate with the local network, although security authentication can extend this.

Caution: Do not send messages until a call to the emberStackStatusHandler() callback informs you that the stack is up.

Rejoining a network is done using a different function: `emberFindAndRejoinNetwork()`. The application may call this function when contact with the network has been lost. The most common usage case is when an end device can no longer communicate with its parent and wishes to find a new one.

The stack will call `emberStackStatusHandler()` to indicate that the network is down, then try to re-establish contact with the network by performing an active scan, choosing a parent, and sending a ZigBee network rejoin request. A second call to the `emberStackStatusHandler()` callback indicates either the success or the failure of the attempt. The process takes approximately 150 milliseconds to complete.

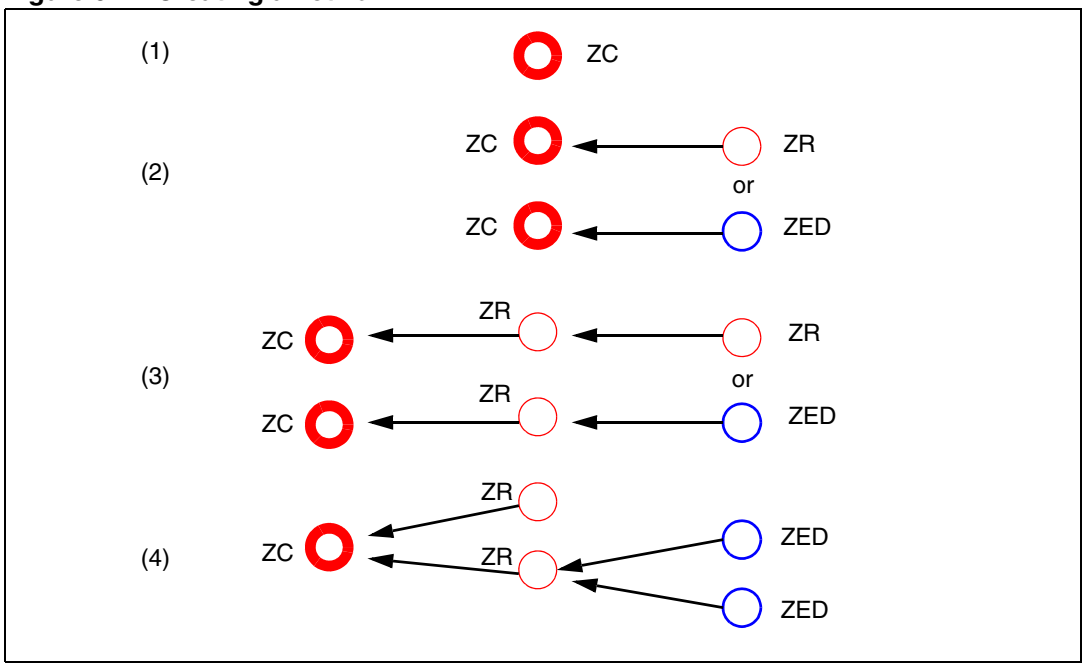
Another case when this function is useful is if the device may have missed a Network Key update and thus no longer has the current Network Key.

Note: `emberFindAndRejoinNetwork()` *replaces the* `emberMobileNodeHasMoved()` *API from EmberZNet 2.x, which used MAC association.*

Creating a network

To create a network, a node must act as a coordinator (ZC) while gathering other devices into the network. The process generally follows a pattern as shown in *Figure 9*:

Figure 9. Creating a network



1. ZC starts the network by choosing a channel and unique two-byte PAN-ID and extended PAN-ID (see *Figure 9*, step 1). This is done by first scanning for a quiet channel by calling `emberStartScan()` with an argument of `EMBER_ENERGY_SCAN`. This identifies which channels are noisiest so that ZC can avoid them. Then, `emberStartScan()` is called again with an argument of `EMBER_ACTIVE_SCAN`. This provides information about PAN IDs already in use and any other coordinators running a conflicting network. (If another coordinator is found, the application could have the two coordinators negotiate the conflict, usually by allowing the second coordinator to join the network as a router.) The ZC application should do whatever it can to avoid PAN ID conflicts. So, the ZC selects a quiet channel and an unused PAN ID and starts the network by calling `emberFormNetwork()`. The argument for this function is the

parameters that define the network. (Refer to the online API documentation for additional information.)

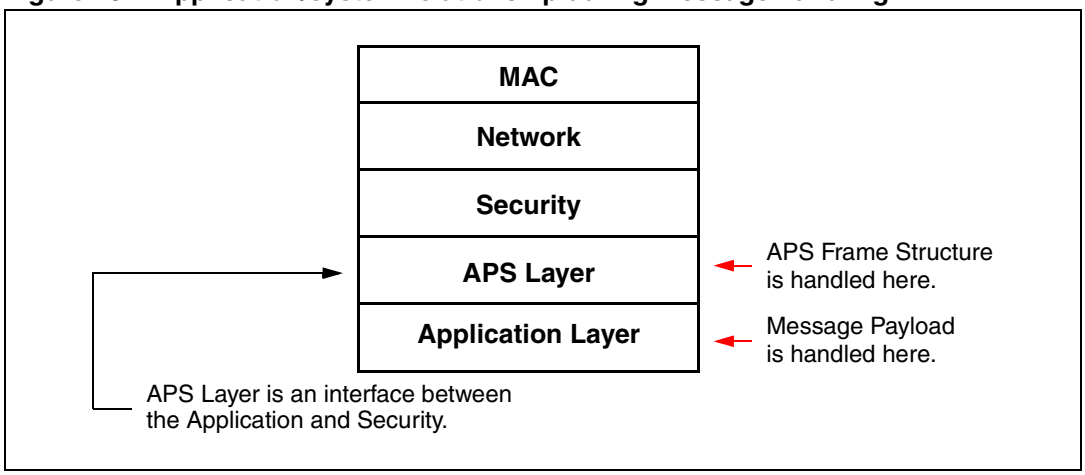
Note: The ZC's application software must, at compile time, specify the required stack profile and application profile for the endpoints implemented. The stack is selected through the `EMBER_STACK_PROFILE` *global definition. This may have an impact on interoperability.*

2. ZR or ZED joins the ZC (see *Figure 9*, step 2).
3. ZR or ZED joins the ZR (see *Figure 9*, step 3).
4. This results in a network with established parent/child relationships between nodes.

Once the new network has been formed, it is necessary to direct the stack to allow others to join the network. This is defined by the function `emberPermitJoining()`. The same function can be used to close the network to joining.

4.3.4 Message handling

Figure 10. Application/system relationship during message handling



There are many details and decisions involved in message handling, but it all simplifies into two major tasks:

- Create a message
- Process incoming messages

The EmberZNet stack software takes care of most of the low level work required in message handling. *Figure 10* 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.

Sending a message

There are three basic types of messages that 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 that must be considered is the maximum size of the message payload originating in your application.

Take a few moments and study the structure of the following API[a] functions:

Table 8. API structures

Function	Description
`emberSendUnicast ( EmberOutgoingMessageType type, int16u indexOrDestination, EmberApsFrame * apsFrame, EmberMessageBuffer message )`	Sends a unicast message as per the ZigBee specification. Parameters: `type` Specifies the outgoing message type. Must be one of `EMBER_OUTGOING_DIRECT`, `EMBER_OUTGOING_VIA_ADDRESS_TABLE`, or `EMBER_OUTGOING_VIA_BINDING`. `indexOrDestination` Depending on the type of addressing used, this is either the `EmberNodeId` of the destination, an index into the address table, or an index into the binding table. `apsFrame` The APS frame which is to be added to the message. `message` Contents of the message.
`emberSendBroadcast ( EmberNodeId destination, EmberApsFrame * apsFrame, int8u radius, EmberMessageBuffer 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`. Parameters: `destination` The destination to which to send the broadcast. This must be one of three ZigBee broadcast addresses. `apsFrame` The APS frame data to be included in the message. `radius` The maximum number of hops the message will be relayed. `message` The actual message to be sent.
`emberSendMulticast ( EmberApsFrame * apsFrame, int8u radius, int8u nonmemberRadius, EmberMessageBuffer 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. Parameters: `apsFrame` The APS frame for the message. The multicast will be sent to the groupId in this frame. `radius` The message will be delivered to all nodes within this number of hops of the sender. A value of zero is converted to `EMBER_MAX_HOPS`. `nonmemberRadius` The number of hops that the message will be forwarded by devices that are not members of the group. A value of 7 or greater is treated as infinite. `message` A message.

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

a. Message Handling functions are grouped in the online API documentation under “EmberZNet Stack API Reference” and the “Sending and Receiving Messages” function group.

In every case illustrated above, there is a message buffer to contain the message. Normally, the application allocates memory for this buffer (as some multiple of 32 bytes). But how big can this buffer be? Or: How big of a message can be sent? The answer is: you don't necessarily know, but you can find out dynamically. 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, there is a return value from the `emberSend`... function. Check the online API documentation for further information.

It should become clear that the task of sending a message is a bit complex, but it is also very consistent. The challenge in designing your application is keeping 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.

Receiving messages

Unlike sending messages, receiving messages is a more open ended process. The application will be notified when a message has been received, but the application must decide what to do with it and how to respond to it.

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 STM32W108, the stack will deal with the mechanics of receiving and storing a message. In all cases, the application must parse the message into its constituent parts and decide what to do with the information. This will vary based on the system design, which only allows us to discuss it in general terms. 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.

When a message is received, the function `emberIncomingMessageHandler()` is a callback invoked by the EmberZNet stack. This function contains the following arguments:

Table 9. emberIncomingMessageHandler() arguments

Function	Description

`emberIncomingMessageHandler ( EmberIncomingMessageType type, EmberApsFrame * apsFrame, EmberMessageBuffer message )`	`type` The type of the incoming message. One of the following:
	`EMBER_INCOMING_UNICAST`
	`EMBER_INCOMING_UNICAST_REPLY`
	`EMBER_INCOMING_MULTICAST`
	`EMBER_INCOMING_MULTICAST_LOOPBACK`
	`EMBER_INCOMING_BROADCAST`
	`EMBER_INCOMING_BROADCAST_LOOPBACK`
	`EMBER_INCOMING_MANY_TO_ONE_ROUTE_REQUEST`
	`apsFrame` The APS frame from the incoming message.
	`message` The message that was sent.

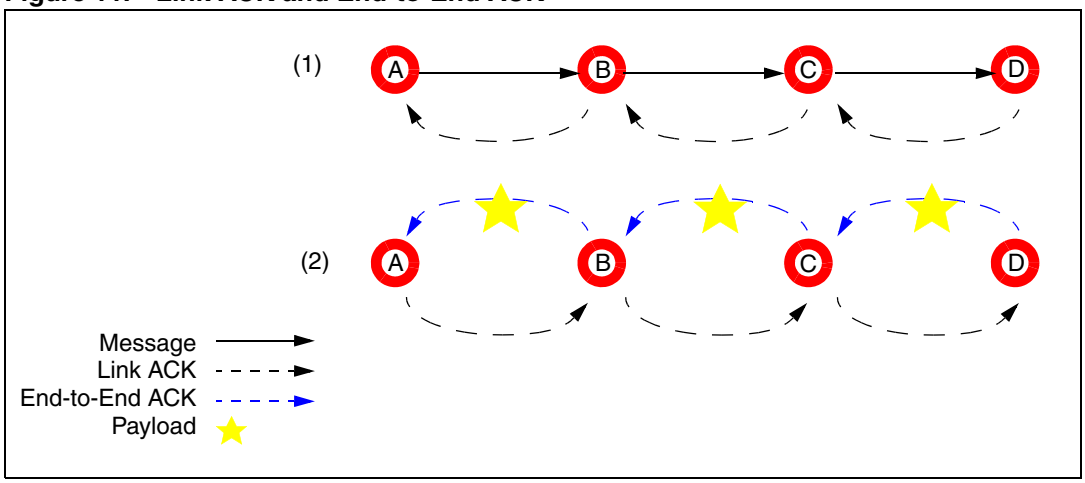
It is clear that there is more than the three message types previously discussed. The others are simply specialized variations of the three basic types.

The application message handling code must deal with all three arguments of `emberIncomingMessageHandler()`. There may be special handling options for certain message types. The message itself must be parsed into its constituent parts and each part reacted to accordingly. This usually involves a switch statement and will vary in detail with every application. The sample applications are a good place to look for detailed examples of incoming message handlers.

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 11* (1) where node A send 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 11* (2)). While adding a payload is possible, it is not compliant with the ZigBee specification.

Figure 11. Link ACK and End-to-End ACK



Note: Additional information on message handling can be found in Section 7.6: ZigBee messaging.

4.3.5 Housekeeping tasks

A variety of housekeeping tasks must be included in the application. Some of these tasks are application dependent, but some are required by every application. These universally required tasks are the subject of this section.

Processor maintenance

In the case of the STM32W108, there is one unique task required of all applications running on that platform. You must call emberTick() on a regular basis. `emberTick()` acts as a link between the stack state machine and the application state machine. When executed, `emberTick()` allows the stack to deal with several tasks that have collected since the last time `emberTick()` was called. A variety of callbacks can result from each call to `emberTick()`, generally to get input from the application for resolving a stack task. As mentioned in Set Up Main Program Loop, `emberTick()` is called in the application's main event loop and after calling `emberInit()`.

The application's main loop must perform the following tasks:

- Manage I/O functions
- Reset the watchdog timer
- Buffer and memory maintenance (you also need to call `emberSerialBufferTick()` if using serial I/O)
- Error Processing
- Debugging features/strategies

Network maintenance

A key group of housekeeping tasks involve network maintenance. All applications must generally deal with the following tasks (based on the complexity of the system/application design):

- Joining a network
- Rejoining a network
- Dealing with interference (that is, being able to adapt to degrading network conditions) (optional)

For an End Device (ZED):

- Loss of connectivity (that is, polling for a lost parent)
- Checking in with parent after emerging from a sleep or hibernation state
- Mobile ZEDs must keep their connection to their parent alive by regularly polling that link or else they will need to call emberRejoinNetwork() whenever they wish to communicate with the network

For a Coordinator (ZC) or Router (ZR):

- Forming a network
- Commissioning a network or accepting a new device on the network
- Dealing with new or replaced nodes

And, if sleepy ZEDs are included in the network:

- ZED power consumption (managing sleeping or waiting end devices)
- Acting as a parent, dealing with ZED alarms, and reacting to child join/leaves

5 Security

5.1 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 12* illustrates, the Security Services Provider block interacts 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. Both Standard and High Security use Trust Centers. MAC-level security is not used by ZigBee.

Figure 12. ZigBee stack architecture

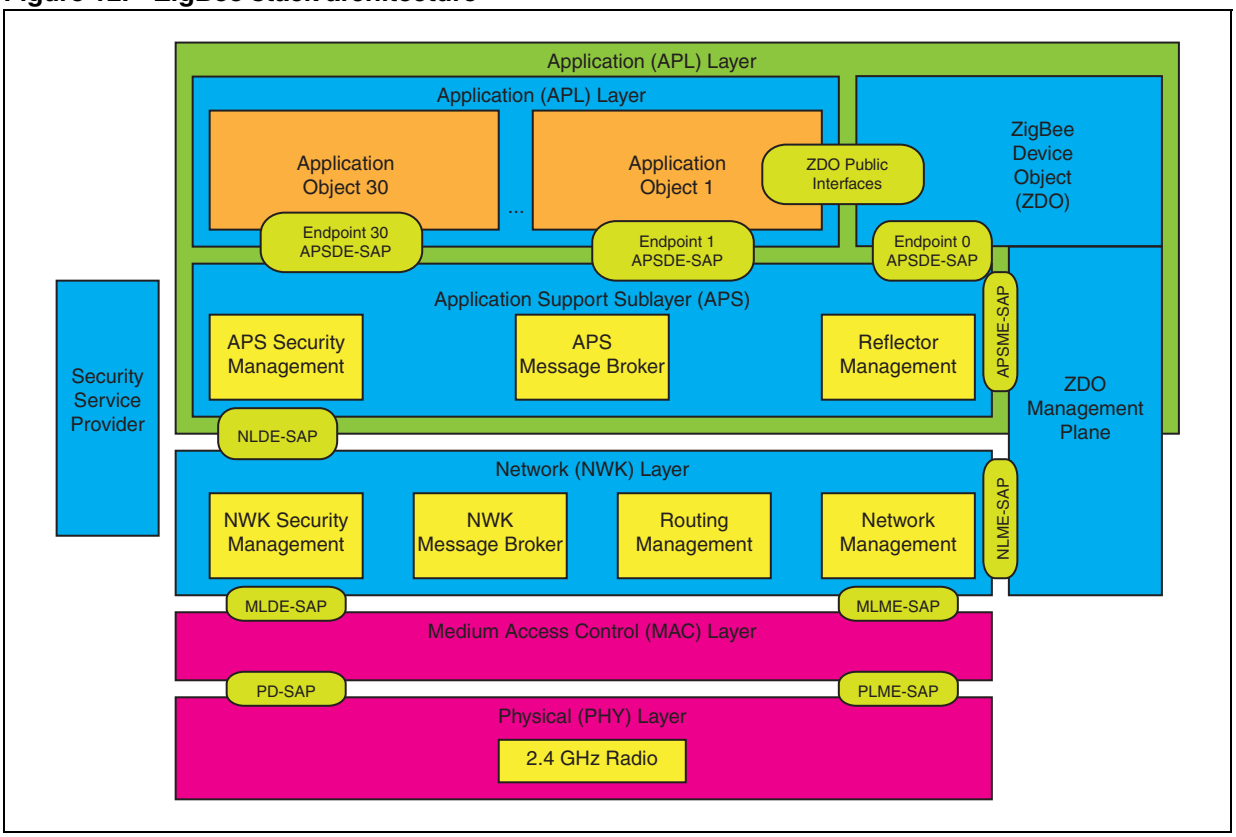


Table 10 describes the different security types in the ZigBee specification. In EmberZNet 3.1, Standard Security replaces Commercial Security implemented in EmberZNet 3.0. Please note that EmberZNet implements Standard Security in two modes: with and without a Trust Center.

Table 10. ZigBee security type

Type	Description	EmberZNet support
Residential	This is the security services provided for the Zigbee 2006 specification. It provides Network Layer security using a Network Key.	Yes(1)
Standard	This is the security services provided under the Zigbee Pro specification. It is Residential Security with a set of optional enhancements. These enhancements include APS Layer Security using Link Keys.	Yes
High	This is an optional security service provided under the ZigBee Pro specification. High security is not supported by EmberZNet.	No
No Security	While it is possible to design an application with no security features, It is not recommended doing so. Such an application is also not ZigBee-compliant.	Yes

1. EmberZNet supports connecting ZigBee end devices using Residential Security to ZigBee Pro networks running Standard Security and vice versa.

Note: High security is not supported by EmberZNet and is not currently used by any application profiles, including Smart Energy. It will not be discussed further in this chapter.

Those already familiar with ZigBee security can jump to *Section 5.4: Implementing security*.

This chapter first examines Network and Application Layer security features. Then the discussion will shift to the types of Standard Security protocols available in EmberZNet. Lastly, coding requirements for implementing security will be reviewed.

5.1.1 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 will always be used. The application has no ability to turn it off or send packets unencrypted. For application controlled security, see *Section 5.1.2: APS Layer Security*.

Network key

Network Security utilizes 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.

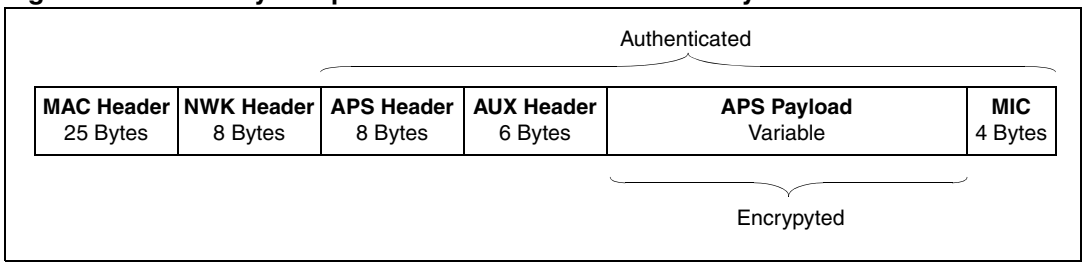
Hop-by-hop security

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 (that is, 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.

Packet security

A packet secured at the network layer is composed of the elements shown in *Figure 13*.

Figure 13. Anatomy of a packet secured at the network layer



Auxiliary header

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

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) and that 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 will hash the message and verify the calculated MIC against the value appended to the message. Alterations to the message will invalidate the MIC and the receiving node will discard the message entirely.

Note: ZigBee uses a 4-byte MIC.

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. Prior to the frame counter hitting its maximum value, the Network Key can be updated. When that occurs, the sequence number is reset back to zero to reflect the use of a different Network Key.

Unencrypted network data

If Network Security is being used, all packets will be secured. The only exception to this 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.

5.1.2 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.

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.

It is possible that both APS and Network Layer encryption may be used to encrypt the contents of a message. In that case APS Layer Security is applied first, then Network Layer Security.

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 prior to sending APS secured data. There are two types of Link Keys: Trust Center Link Keys and Application Link Keys.

Trust center link keys

This is a special Link Key in which one of the partner devices is the Trust Center. This key is used by the stack to send and receive APS Command messages to and from the Trust Center. It may also be used by the application to send APS-encrypted data messages.

Standard Security does not require Trust Center Link Keys, but devices may request one after joining. It is highly recommended using Trust Center Link Keys. They are required for any device that wishes to rejoin a network of which it was previously a member.

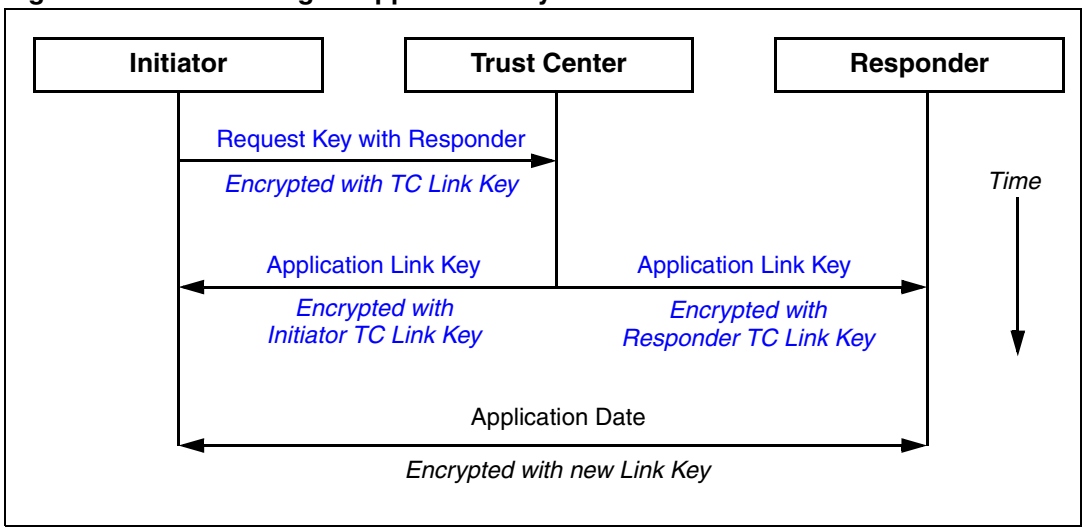
Application link keys

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.

Devices 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 shown in *Figure 14*.

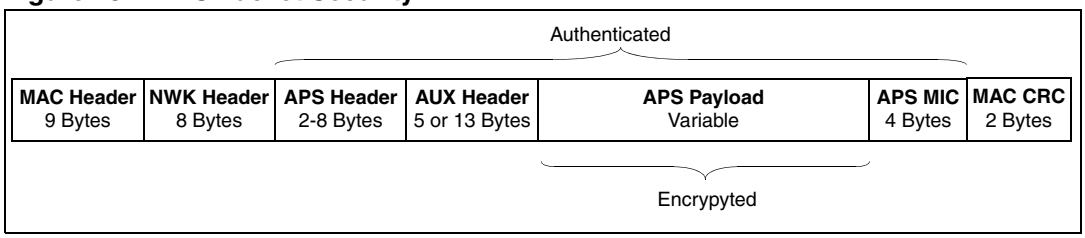
Figure 14. Establishing an application key



APS packet security

A packet secured at the APS Layer is composed of the elements shown in *Figure 15*.

Figure 15. APS Packet Security



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.

5.2 Residential security

5.2.1 Overview

Residential Security is the only security available to ZigBee 2006 devices. It utilizes Network Security to encrypt all traffic in a ZigBee Network. All devices in the Network must be informed of the Network Key in one of two ways:

1. Preconfiguring the Network key before joining the network.
2. Sending the key in the clear to the joining device. This one-time event poses a risk but may be acceptable, depending on the application.

Note: ZigBee uses a Key of all zeroes as a special indicator in Residential Security. Therefore, it is not a valid Network Key.

EmberZNet reserves a key of all F's as a special value, which cannot be used when setting up keys

5.2.2 Trust center

Authentication 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 to allow or disallow 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 will relay in the clear 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.

5.2.3 Residential security keys

The only key used in Residential Security is the Network Key. 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.

5.3 Standard security

5.3.1 Overview

Standard Security is similar to Residential Security but includes several optional enhancements to further increase security within the Network. It is backward-compatible

with Residential Security and allows ZigBee 2006 devices to communicate securely on a ZigBee Pro network. Standard Security is only available to ZigBee Pro devices.

Standard Security utilizes Network and Link Keys to encrypt data at the Network and Application layers, respectively. The Application Layer Security allows the Trust Center to securely transport the Network Key to joining or rejoining nodes, and it optionally allows applications to add further security to their messages. Network Layer Security is used to secure all traffic sent on a Zigbee Network.

One of the other significant changes in Standard Security is the addition of end-to-end security at the Application Layer to supplement the network-wide security (see *Section 5.1.2: 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 that devices must adhere to.

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 when 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.

5.3.2 Trust center

Standard Security also relies on a Trust Center to authenticate devices joining the network. The Trust Center acts much the same in Residential Security as in Standard Security, but it also has the added responsibilities of distributing and managing Trust Center Link Keys and responding to requests for Application Link Keys.

5.3.3 Standard security keys

Standard Security uses most of the same keys as in Residential Security, but also defines additional 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.

Network key

This is the network-wide key used to secure transmissions at the Network Layer. It is utilized the same way in Residential Security as in Standard Security. How the Network Key is transported and delivered to nodes that are part of the network is the main difference in Standard Security.

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.

Application link keys

Standard Security supports devices establishing Application Link Keys with other devices. These are separate from the Trust Center Link Key and not required for normal operation. These application link keys are used for APS level encryption between two devices in the network.

These application link keys 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.

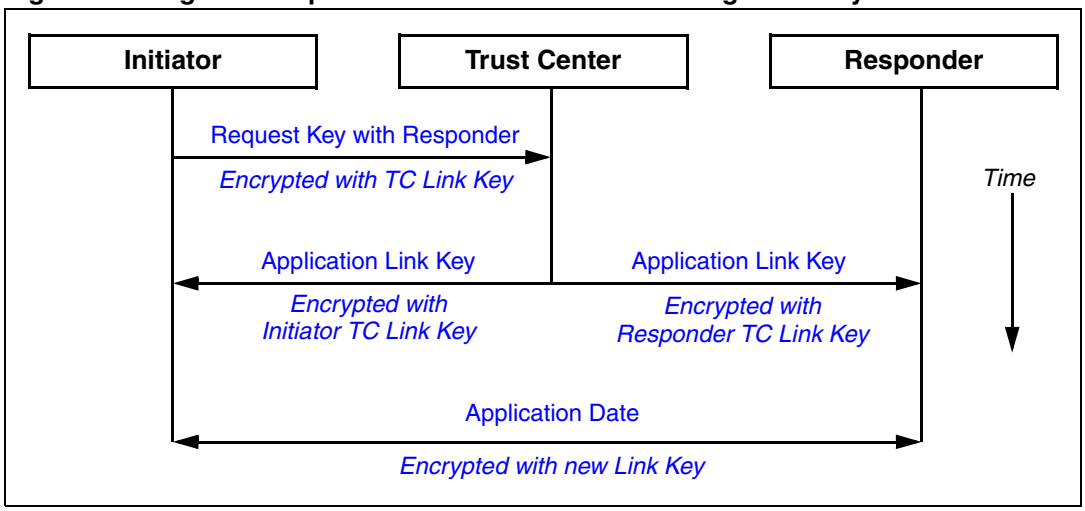
Application Link Keys can be established in one of two ways:

1. Manual configuration by the application specifying the key associated with a destination device.
2. Requesting 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.

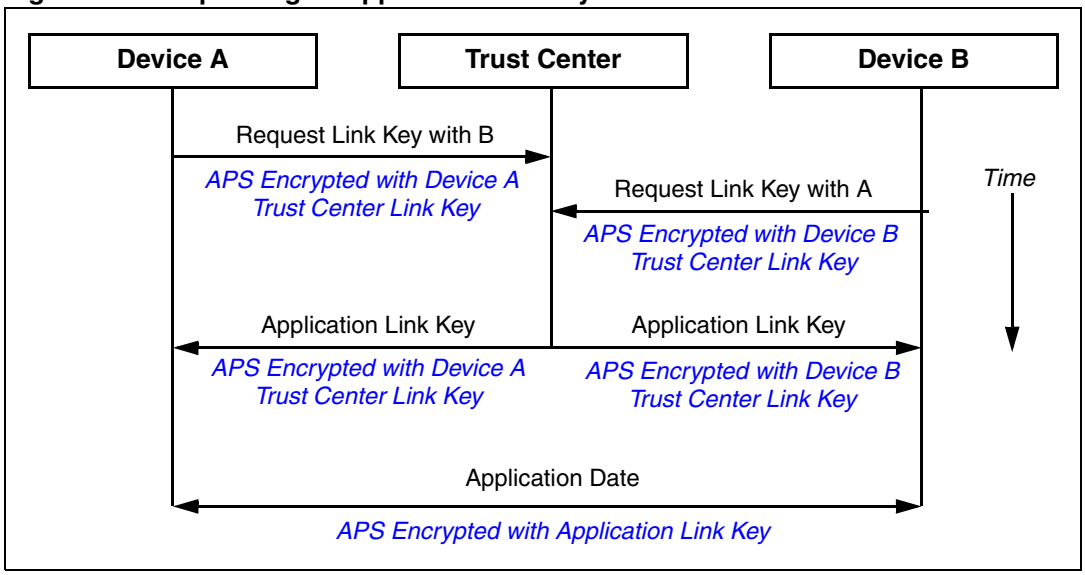
Application Link Keys can also be established using the Trust Center. The EmberZNet Stack supports two ways to configure how this is done. The first is the ZigBee compliant method, shown in *Figure 16*, where one device requests an Application Link Key with another device by contacting the Trust Center. The Trust Center then immediately responds and sends a randomly generated Application Link Key back to the requesting device, as well as 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.

Figure 16. ZigBee-compliant mechanism for establishing a link key



The second method, shown in *Figure 17*, is not ZigBee-compliant, but it helps insure that the device will be online and able to receive an Application Link Key. In this case, both devices are required to request an Application Link Key from the Trust Center. The Trust Center will store 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 their own Application Link Key request with the first device as its partner. If that occurs, then the Trust Center will generate a random Application Link Key and send 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 Application Link Key.

Figure 17. Requesting an application link key with another device on the network



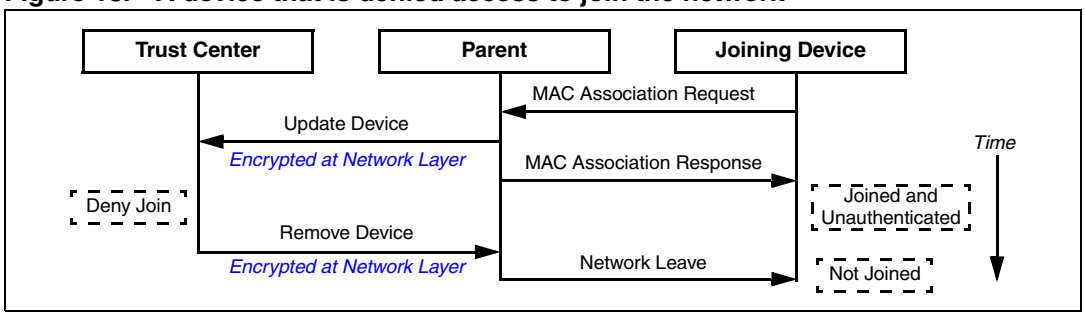
EmberZNet supports a configurable table for storing Application Link Keys. See *Section 5.4.4: Common security configurations* for more information.

5.3.4 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, see *Figure 18*. If the device is allowed to join, the Trust Center's behavior depends upon whether the device has a preconfigured Link key.

Figure 18. A device that is denied access to join the network



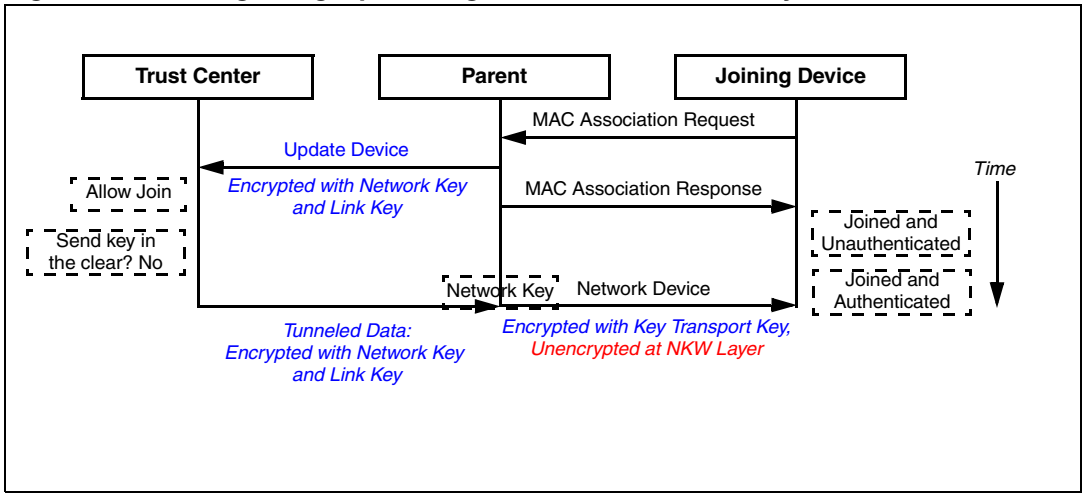
Preconfigured link keys

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 (e.g., the joining node's EUI64 Address), or unique, randomly generated keys for each device.

To allow a device onto the network, the Trust Center transmits the Network Key encrypted with the device's preconfigured Link Key.

Figure 19. Joining using a preconfigured trust center link key



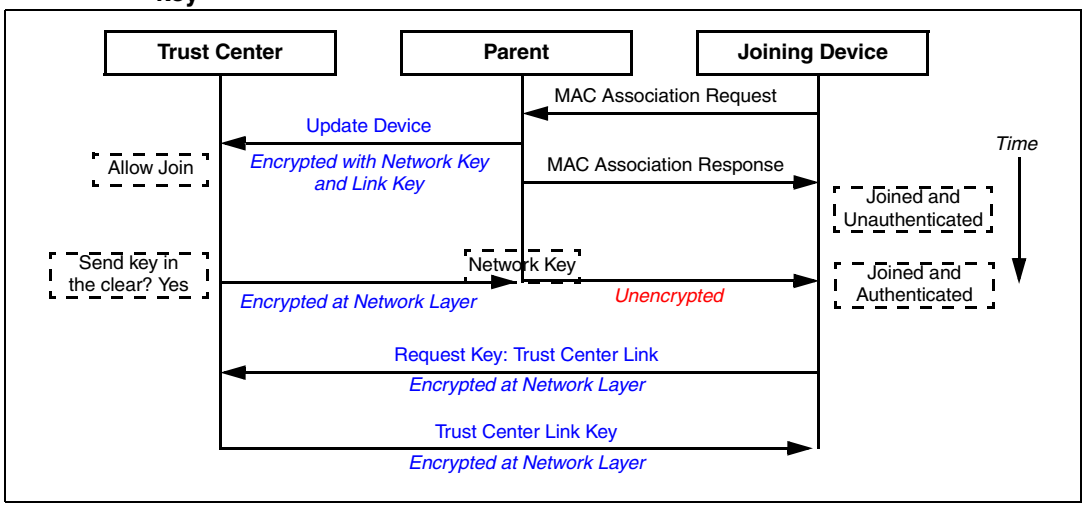
No Preconfigured keys

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.

Requesting a link key

One of the advantages with Standard Security is that, even if a preconfigured Link Key is not required for joining (that is, 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 in *Figure 20*.

Figure 20. 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.

5.3.5 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.

Alternatively, a Trust Center may want 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 insure 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 will 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 or it 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.

5.3.6 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.

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.

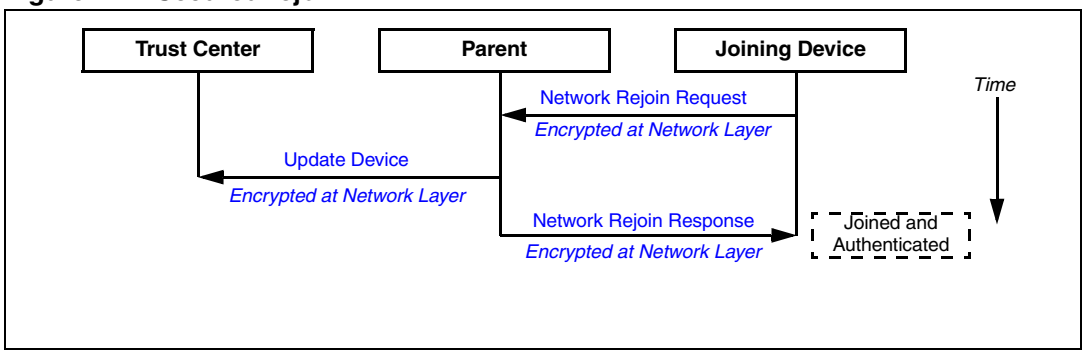
Therefore, a device has two choices when rejoining: a Secured Rejoin or an Unsecured Rejoin.

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 21*, 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 21. Secured rejoin



If the Secured Rejoin fails and the device is using Standard Security, the application can try an Unsecured Rejoin. If the device has neither the current Network Key nor a Trust Center Link Key, it will have to perform a join.

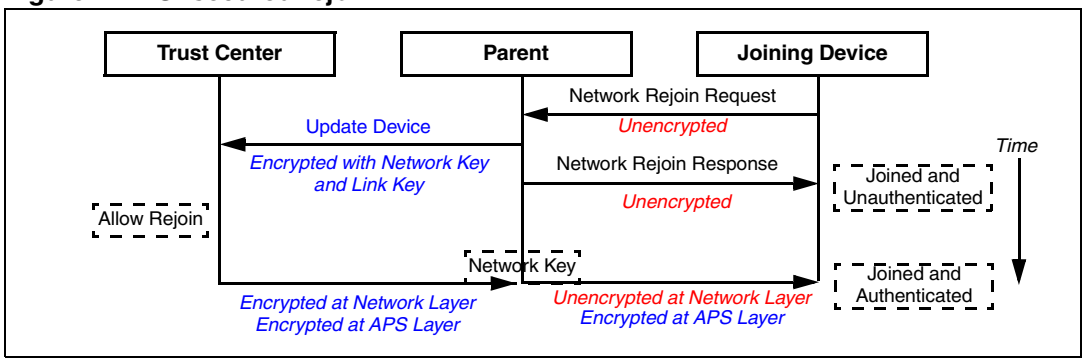
Unsecured rejoining

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 22*, 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.

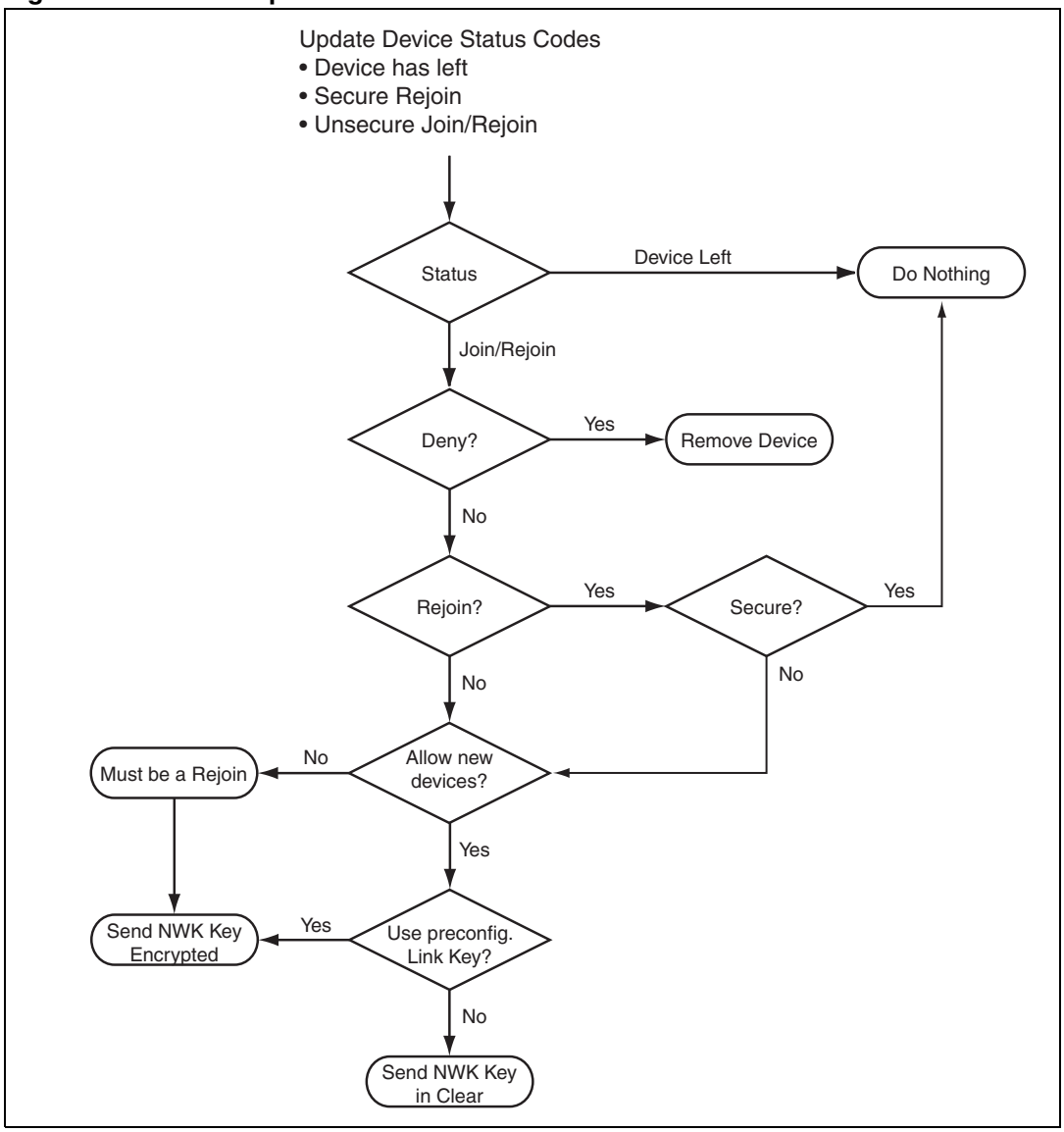
Figure 22. Unsecured rejoin



5.3.7 Summary

Figure 23 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 23. Decision process for the trust center



The Trust Center has the choice of deciding 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 (network is being commissioned).

When new devices join, the decision of whether a device should have a preconfigured key is left to the Trust Center. The joining devices have no ability to inform the Trust Center via the ZigBee protocol about whether or not they have a preconfigured key.

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 initially developing applications or when commissioning a network, the Trust Center may not be reachable, and thus devices will not be able to join.

The EmberZNet Stack allows a network to utilize 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: Distributed Trust Center Mode" was previously known as "No Trust Center Mode."

In this mode, all devices utilize 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 Network Key is sent 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: It is not possible to have your application certified as ZigBee-compliant without implementing a Trust Center.

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 via a Network Key Update that it is the new Trust Center.

5.3.8 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 utilize an Address Cache for security, in order to maintain a mapping of IEEE Address to Short ID.

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 stack will automatically add a route to the Trust Center through their parent (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 will 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.

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.A Source Route Library is provided by EmberZNet which manages a Source Route Table, and works with a Trust Center.

Note: The Source Route Table on the Host cannot be used for routing security messages sent to devices joining or rejoining the network.

Trust center address cache

The Trust Center must maintain a mapping of IEEE address to Short ID. This is necessary in order to properly decrypt APS encrypted messages.

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.

Sample code has been provided for the Trust Center Address Cache on the STM32W108.

5.4 Implementing security

Security is implemented by how the stack software is configured and how the message packets are configured.

Note: Complete API documentation can be found with the online tools supplied with EmberZNet.

5.4.1 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. EmberZNet 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.

5.4.2 Security for forming and joining a network

For devices to form or join a ZigBee Pro Network utilizing Standard Security, they must call emberSetInitialSecurityState(…) prior to calling `emberFormNetwork(...)` or `emberJoinNetwork(...)`.

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

All initial security parameters are set via 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 prior to forming or joining. Once security is set up and a device is joined into the network, it will persistently store 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 will be reset to zero.

Initial security state structure

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

```
typedef struct {
  // This bitmask controls what security features are used, and
  // whether the preconfiguredKey and networkKey elements below
  // contain valid keys.
  EmberSecurityBitmask bitmask;
  // This is the preconfiguredKey that will be used to send
  // encrypted security data to the joining device.
  // For the trust center operating in standard security mode this
  // will contain the link key and must be set, regardless of
  // whether the joining device has the key preconfigure.
  EmberKeyData preconfiguredKey;
  // This key is only used by the trust center when forming the
  // network.  It is ignored on joining.
  EmberKeyData networkKey;
  // This is the sequence number associated with the network key.
  // It is used when the trust center forms the network.  It is
  // not used by joining devices.
  int8u nwkKeySequenceNumber;
} EmberInitialSecurityState;
```

Initial security bitmask

Table 11 describes the different settings used for security and what devices may set them.

Table 11. Initial security bitmask

Bits	Name	May be set by:	Description
1	EMBER_DISTRIBUTED_TRUST_CENTER_MODE	Device that forms network	This controls whether the device will create a network operating with (0) or without (1) a Trust Center. All devices that join the network will inherit this setting from their parent.
2	EMBER_GLOBAL_LINK_KEY	Trust Center	This controls whether the Trust Center is using the same Link Key for all devices (1), or separate Link Keys for each device (0). This must be set when operating in Distributed Trust Center Mode.
3	EMBER_PRECONFIGURED_NETWORK_KEY_MODE	Trust Center	This controls whether the Trust Center utilizes 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	This bit determines whether the device has set a value in the preconfiguredTrustCenterEui64 field in the EmberInitalSecurityState structure. If the value is set, that field will be taken as the EUI64 of the trust center in the network. Normally this field should NOT be set, because a joining device will learn the EUI64 of the trust center when it receives the network key. However, in the case of a device that is being commissioned to join an existing network without sending any over-the-air messages, it must also set this bit and populate the field appropriately.
7,2	EMBER_TRUST_CENTER_USES_HASHED_LINK_KEY	Trust Center	This 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 insure that devices do not share the same Trust Center Link Key throughout the network.

Table 11. Initial security bitmask (continued)

Bits	Name	May be set by:	Description
8	EMBER_HAVE_PRECONFIGURED_KEY	Forming or Joining Devices	This controls whether or not there is valid data in the Preconfigured Key element of the EmberInitialSecurityState structure. If set (1), the stack will record the key in persistent storage. If not set (0), that parameter will be ignored. This must be set for the device that forms the network, and is separate from the Trust Center decision to use preconfigured keys. For joining devices, this controls whether or not the device has a preconfigured key. In EmberZNet 3.1 this preconfigured key is the Trust Center Link Key.
9	EMBER_HAVE_NETWORK_KEY	Forming Device	This controls whether or not there is valid data in the Network Key element of the EmberInitialSecurityState structure. If set (1), the stack will record the key in persistent storage. If not set (0), that parameter will be ignored. This must be set prior to forming the network. It is not needed for joining.
10	EMBER_GET_LINK_KEY_WHEN_JOINING	Joining Devices	This controls whether the joining device will request a Link Key after it receives the Network Key. If set (1), the joining will not be successful until a request is sent and a response is received. If unset (0), the device will not request a Link Key.
11	EMBER_REQUIRED_ENCRYPTED_KEY	Joining Devices	This controls whether a joining device with a preconfigured Link Key will accept a Network Key sent in the clear. If set (1), a network key sent in the clear will be rejected. If unset (0), the device will accept 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	This denotes whether the device should NOT reset its outgoing NWK and APS frame counters prior to 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 will use 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().

Table 11. Initial security bitmask (continued)

Bits	Name	May be set by:	Description
13	EMBER_GET_PRECONFIGURED_KEY_FROM_INSTALL_CODE	Joining Devices	Configured 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() will fail.
14-15	Reserved		

5.4.3 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.

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 Residential Security, and since Standard Security is compatible with Residential Security, this key value is prohibited. Attempts to call emberSetInitialSecurityState(...) with EMBER_HAVE_NETWORK_KEY and a Network Key of all zeros will fail.

Frame counters

Messages that are encrypted use Frame Counters to prevent messages from being replayed into the network. There are two types of Frame Counters: 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.

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.

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 are the types of Replay Protection Offered:

- 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 will cause the device to forget the incoming Frame Counter 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 will re-synchronize the Frame Counter with its partner using a challenge-response mechanism. This is not available in EmberZNet 3.1. The application may implement a mechanism that enables this behavior.

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 an inordinate 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

Global trust center link key

A Global Trust Center Link Key is the simplest option. 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.

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 will be 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 will not keep track of Incoming APS Frame Counters for individual devices which 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 at http://csrc.nist.gov/groups/ST/toolkit/message_auth.html

Unique trust center link keys

The Trust Center does have the option of utilizing totally unique Trust Center Link Keys for a small set of devices in the network. In this case the Trust Center will store the Incoming Frame Counters associated with those Link Keys and thereby have replay protection for APS Encrypted messages it receives from those devices.

Unique Trust Center Link Keys requires the Link Key Security Library. The Trust Center must setup a key for the device prior to that device joining, regardless of whether the key is being sent in-the-clear or is preconfigured. Each entry in the table consumes both Flash and RAM. For more information on the Flash and RAM requirements see *Section 5.4.8: Link key libraries*.

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 will first consult the Link Keys Table for a Unique Trust Center Link Key associated with the sending/receiving device. If none exists, then it will fall 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.

Summary of Replay Protection

Table 12 provides a summary of the replay protection provided under the different security configurations supported.

Table 12. Replay protection level

Layer	Device type	Key type	Incoming frame counter protection
APS	Trust Center	Global Trust Center 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	Strong
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.

5.4.4 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.

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 via non-Zigbee wireless methods where the key is sent via 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 will learn about this when it joins the network. It does not need to set anything up when joining the network since it has no control whether or not the network is operating in that state.

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

1. `emberNetworkInit(...)`
2. If not joined (`EMBER_NOT_JOINED`)...
   a) `emberSetInitialSecurityState(...)`
   b) `emberJoinNetwork(...)`

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

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 will reject that key and fail 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 incorrect devices.

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 will be 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 will request a Trust Center Link Key. If a Trust Center Link Key is not received then the join will fail with the error code `EMBER_NO_LINK_KEY_RECEIVED`.

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.

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 that 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. Will ZigBee 2006 devices need to be supported which have a preconfigured Network Key?

The Trust Center must always set up a Network Key. It should setup a Link Key prior to 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, device 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 5.4.6: 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 in the network (`EMBER_NOT_JOINED`)…
   a) `emberSetInitialSecurityState(…)`
   b) `emberFormNetwork(…)`

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

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 and Link Key so that devices on the network can obtain both. Every single Router that joins the network will mimic some of the behaviors of the Trust Center. Devices that join to a router will 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 via the Trust Center Join Handler.

Global trust center link keys

This configuration is the simplest Trust Center configuration. All devices will 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);
```

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 will be 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);
```

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.

For those joining devices the Trust Center wants to configure a Link Key, call Link Key Table API to add a Link Key.

1. `emberAddOrUpdateKeyEntry(address, TRUE, keyData)`
   a) Join the device(s) to the Network.
2. If the device joins before the Unique Trust Center Link Key is setup then the Trust Center will try to use the Global or Hashed Link Key.

5.4.5 Error codes specific to security

A number of new error codes may be returned by the stack that pertain to security. *Table 13* describes the error codes and when they may be received.

Table 13. 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 prior to 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	This means that 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.

5.4.6 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 (e.g., button presses, 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 and the device supports Residential Security (EMBER_STANDARD_SECURITY_SECURED_REJOIN)
3. An Unsecure Join was performed and the device supports Residential Security (EMBER_STANDARD_SECURITY_UNSECURED_JOIN).

4. An Unsecure Rejoin was performed and the device supports Residential Security (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 Unsecure Join and Unsecure 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).

5.4.7 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 what 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)
```

Current security state structure

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

Current security bitmask

The bitmask of the current security settings is shown in *Table 14*.

Table 14. Bitmask of current security settings

Bits	Name	Information Applies to	Description
1	EMBER_DISTRIBUTED_TRUST_CENTER_MODE	All Devices	This 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 will not contain any valid data.
2	EMBER_GLOBAL_LINK_KEY	Trust Center only	This 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	This specifies whether the device has a Trust Center Link Key. If set (1), it indicates the device does have a Trust Center Link Key. If not set (0), no Trust Center Link Key is available.
5-6	Reserved		
2,7	EMBER_TRUST_CENTER_USES_HASHED_LINK_KEY	Trust Center	If set (1), this indicates that the Trust Center is using a Hashed Link Key to derive individual link keys from the preconfigured (Root key.
8-15	Reserved		

Obtaining the security keys

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

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

The call will fetch the requested key type and copy 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 part of High Security and therefore not supported in EmberZNet.

- Trust Center Master Key (EMBER_TRUST_CENTER_MASTER_KEY)
- Application Master Key (EMBER_APPLICATION_MASTER_KEY)

Key data structure

The EmberKeyStructure contains Key data as well the associated information about the Key. The fields in the Key structure will 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;
```

Key structure bitmask

Table 15 describes the Key structure bitmask.

Table 15. 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	

5.4.8 Link key libraries

Normally a device can only store one link key, the Trust Center Link Key. However for those devices that wish to store more than one Link Key (including Application Link Keys), there is the Link Key Library. The Link Key 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 Keys Library utilizes a table stored in flash and RAM to keep track of the keys and their associated data. The table's size is configurable via 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:

- Key data (16-bytes)
- 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:

- 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 from the Trust Center and thus each device type (Trust Centers and non Trust Centers) can decide independently whether to use the Library.

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 there is a finite amount of devices on the network and the Trust Center has enough flash and RAM to store unique Link Keys for every device, it is recommended to provide a global or hashed Trust Center Link Key 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. Prior to that device joining the network the Trust Center should setup the Link Key in the table by calling into the Link Key Table API. The joining device does not need to utilize the Link Key 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 will utilize the Link Key setup in the table for that device.

When utilizing a mix of Hashed or Global Link Keys, and unique Link Key entries in the Key Table, the Trust Center will always consult 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 will fall 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.

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 each agreeing upon 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 may 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 utilize 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 will be returned via the `emberKeyEstablishmentHandler(...)`.

The length of time a requesting device waits before it times out the request is configurable via the EMBER_REQUEST_KEY_TIMEOUT configuration item. The default value is 2 minutes, but it can be set to a value between one and ten minutes. On a normal node this controls how long it will wait before returning a value of EMBER_KEY_ESTABLISHMENT_TIMEOUT via the emberRequestKeyEstablishmentHandler(…).

On the Trust Center this configuration value controls how long the Trust Center will wait 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.

Link key table API

When new keys are sent to the device by the Trust Center, the stack will automatically update the table and notify the application. Otherwise it is up to the application to manage the Link Key Table. The stack will never delete an entry from the table.

The following are the API calls pertaining to the Link Key Library:

Table 16. API calls pertaining to the link key library

Link key table function	Description
`boolean emberSetKeyTableEntry(int8u index, EmberEUI64 address, boolean linkKey, EmberKeyData* keyData)`	This function 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 0's or all F's, the operation will fail and FALSE is returned.
`boolean emberGetKeyTableEntry(int8u index, EmberKeyStruct* result)`	This function 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)`	This function 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 will fail and the function returns FALSE.
`int8u emberFindKeyTableEntry(EmberEUI64 address, boolean linkKey)`	This function 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)`	This function erases the data in the key table at the specified index. If the index is out of range then the operation will fail and FALSE is returned.

Table 16. API calls pertaining to the link key library (continued)

Link key table function	Description
`EmberStatus emberRequestLinkKey(EmberEUI64 partner)`	This function is valid only for non Trust Center devices. It 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 will be immediately sent back to the device. If the partner address is not the Trust Center, then the Trust Center will store the request until the partner device also requests a key. Then an Application Link key will 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 will be returned via the `emberKeyEstablishmentHandler()`. If a key already exists, then on successful receipt of the new key the old key will be replaced.
`void emberKeyEstablishmentHandler(EmberKeyStatus status)`	This function is 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 does keep track of whether the key is a Master Key or Link Key, only the Link Key type is used by the stack. EmberZNet does not support using Master Keys to derive Link Keys.

5.4.9 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 there are several advantages to using APS Encryption on top of Network Layer encryption.

1. Only the sending and receiving devices should have the Link Key used for encrypting and decrypting messages.
   a) Exception: Messages sent to or from a Trust Center where the Trust Center is utilizing 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 setup via 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 setup 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 will be 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 will be 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 will be rejected silently if they do not.

APS Encryption requires additional overhead and will consume more of the message payload. APS Encryption uses 9-bytes of the payload (5-bytes for a Security Header and a 4-byte MIC).

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 it wishes to send/receive APS encrypted messages from.

If a device cannot determine the long address from the short address of an APS encrypted message, encryption will fail and it will be 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

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 for its own use.

The Trust Center's Long Address is stored in a joining device's tokens. The Short Address is always that of the coordinator, 0x0000.

5.4.10 Updating and switching the network key

Changing the Network Key is a two-step process that requires the Trust Center to first broadcast a copy of the next Network Key, and then tell 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 will 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, this call is made 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.

Sleepy and mobile end devices

It is possible that 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.

EmberZNet 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 Unsecure Rejoin upon receiving that message. Because it is possible that the Rejoin may fail due to any number of networking issues (for example, a bad link), the application may need to make a call to rejoin the network on its own.

Notification of a switch to a new network key

All devices have the ability to 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 via a call to `emberGetKey(...)`.

5.4.11 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 will cause 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 will take 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.

5.4.12 Transitioning from distributed trust center mode to trust center

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 5.4.8: Link key libraries*.

The new Trust Center must also take into consideration the requirements presented in *Section 5.3.8: Additional requirements for a trust center*.

6 Tools

6.1 Introduction

As with most embedded development technologies, a set of tools is available for allowing you (the developer) to create a product using STMicroelectronics's ZigBee products. Each STM32W108 chip 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 created tools from scratch.

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

- EmberZNet stack software
- Compiler toolchain
- Network debugging toolchain

The actual toolchain that you will use is device-dependent. *Table 17* summarizes the major tools for each device.

Table 17. Toolchain summary

EmberZNet Stack Software	Compiler	Network Debugger
STM32W108		
Libraries, HAL source, API Documentation, Sample Applications	IAR EWARM: IDE Compiler, Online Help; Debugger (device level); Document Library	Online Help + Network Analyzer tool (delivered with the STM32W108 Kits)

Utilities

Part of the toolchain consists of a collection of utilities. These may include:

- Bootloaders
- Programming support tools
- Token utility

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

We will now take a closer look at the most important elements of the toolchain.

6.2 EmberZNet stack software

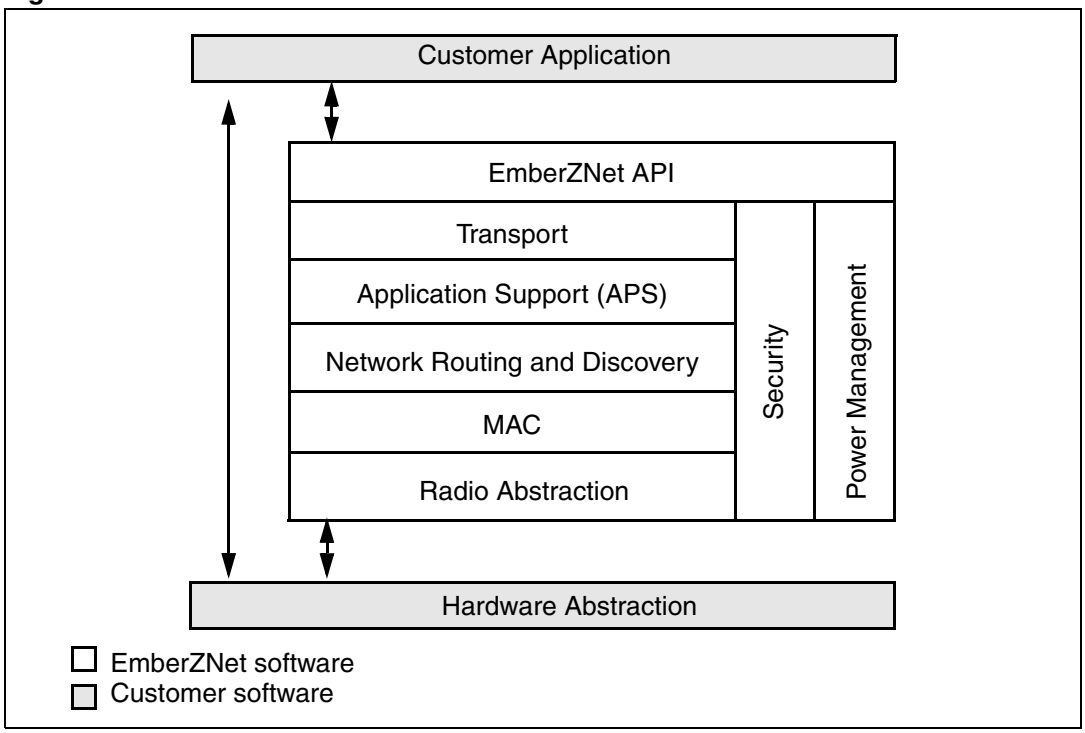
The EmberZNet Stack Software is a collection of libraries, source code, tools, sample applications, and product documentation. The latest version of EmberZNet at the time of this writing is release 4.x, also known as EmberZNet Pro. Most of this manual deals with how to use EmberZNet 4.x.

The EmberZNet 4.x release supports mesh networks because of their increased flexibility and reliability. Consequently, all EmberZNet application must be linked with the mesh library. *Figure 24* illustrates how customer and EmberZNet software interact.

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

- EmberZNet Stack API Guide (HTML format)
- EmberZNet HAL API Guide (HTML format)
- EmberZNet Application Utilities API Guide (HTML format)
- EmberZNet sample applications (C files with many explanatory comments)
- STMicroelectronics web site (http://www.st.com/mcu, STM32W section)

Figure 24. EmberZNet and customer software interaction



6.3 Compiler toolchain

The STM32W108 uses an ARM® Cortex™-M3 processor and an ARM toolchain. This toolchain provides capabilities that include a compiler, linker, debugger, sample applications and user documentation.

6.4 Peripheral drivers

Embedded source code for drivers of peripherals such as the serial controller and analog-to-digital converter (ADC) are provided in C. These drivers let you incorporate standard functionality in custom applications. For more information on these drivers, see the online document Hardware Abstraction Layer Interface Guide.

6.5 Bootloaders

Bootloading is discussed in detail in *Section 8*.

6.5.1 Standalone bootloader

EmberZNet's over-the-air standalone bootloader lets the system upgrade its own software via an RS-232 serial link or a single- or multi-hop radio connection. Using an over-the-air bootloader can speed up the development process. Furthermore, you can easily upgrade deployed nodes that have over-the-air bootload capability.

6.5.2 Application bootloader

The application bootloader lets you upgrade stack or application software via an RS-232 serial link, or over the air via a single-hop or multi-hop radio connection. Using an over-the-air bootloader can speed up the development process. Furthermore, you can easily upgrade deployed nodes that have over-the-air bootload capability.

6.6 Node test

The nodetest application provides 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 about the nodetest application, see the following:

Table 18. Nodetest application notes

Document	Device
Application Note: Bringing Up Custom Nodes	STM32W108

Most customers have standard product manufacturing test flows, but some do not incorporate RF testing. To address this issue, please see Application Note Manufacturing Test Guidelines for STM32W108 SoC Platforms. 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.

6.7 Utilities

6.7.1 Token utility

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 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*.

6.7.2 Hex file utilities

A set of tools for manipulating hex files is also available:

Table 19. List of hex file tools

Tool	Description
`em3xx_load`	These utilities use a command line (DOS console) application that can be used to program the flash memory space of the STM32W108 via jtag/swo interfaces.
`em3xx_convert`	These utilities are intended for use in converting IAR .s37 application files into * .ebl bootload format or the Intel Hex format (.hex).

In addition to the representation of the application, you have the option to include a representation of the application bootloader of the customer manufacturing tokens.

Please refer to the EmberZNet Utilities Guide for detailed information.

7 Advanced design considerations

This chapter discusses several advanced design issues that affect the use of EmberZNet.

7.1 Aggregation

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.

Aggregation is not available in the ZigBee stack.

Source-route aggregation allows the gateway to initiate control messages to the other nodes at any time.

7.1.1 Background

Early in ZigBee's 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. Sometimes the term "aggregation" is used to refer to this pattern and the term "aggregator" for the gateway node. Our original networking solution was implemented in EmberNet and worked loosely as follows.

The system had to scale to hundreds of nodes, subject to our tight bandwidth and memory constraints. To achieve the bandwidth scaling, the system required only the gateway to perform broadcasts for discovery purposes, while all other devices communicated only via unicast datagrams with the gateway. To achieve the memory scaling, we used source routing in the outbound direction, so only the gateway needed to have a large routing table; all other nodes required only one route table entry per gateway.

The gateway was responsible for keeping a fresh gradient to itself (that is routes from all nodes in the network to itself) by periodically sending out a gradient establishment broadcast. It was also responsible for announcing its presence at the application layer to the other nodes. In EmberNet, this was all possible with one broadcast because we allowed piggybacking of data on route requests.

Devices wishing to communicate with the gateway would create an aggregation binding in their binding table, supplying the short ID of the gateway obtained via the announcement, and happily start sending datagrams away as usual. Datagram replies were source routed back out.

Another requirement that immediately came up was the ability for the heavy storage and processing at the gateway to take place on a Linux box communicating to the gateway's stack via serial. Thus was invented RNAP (Remote Node Access Protocol) and its variegated spawn. RNAP was actually a rehash of an earlier serial protocol imaginitively named ESP (EmberZNet Serial Protocol). Based on our experiences with these various approaches to aggregation, we decided to propose the source-route based solution to ZigBee. It was well received and was incorporated into the specification.

How it works

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

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.

Key aggregation-related APIs

The application of the concentrator uses the following new API call to establish the inbound routes, typically on a periodic basis:

```
EmberStatus emberSendManyToOneRouteRequest(int16u concentratorType,
int8u radius);
```

The `concentratorType` is `EMBER_HIGH_RAM_CONCENTRATOR` or `EMBER_LOW_RAM_CONCENTRATOR`.

- For a High Ram Concentrator, nodes send in route records only until they hear a source routed message from the concentrator, or until a new many-to-one discovery happens.
- For a Low Ram Concentrator, route records are sent prior to every APS message.

Devices wishing to communicate with the concentrator should create an address table entry including the short address of the concentrator. The application should avoid initiating address discovery or other kinds of broadcasts to the concentrator for scalability. Instead, the necessary information should be obtained via broadcasts or multicasts from the concentrator. Also, when sending APS unicasts to the concentrator, the discover route option should be off. If using the binding table rather than the address table, the binding should be of type `EMBER_AGGREGATION_BINDING`, which tells the stack not to initiate route or address discovery for that binding.

From the application's point of view, one of the key aspects of the API is the need to manage the source route information on the concentrator. By defining `EMBER_APPLICATION_USES_SOURCE_ROUTING` in the configuration header, the following two stubbed-out callbacks are exposed to the application:

```
/** @description Reports the arrival of a route record command frame
 * to the application. The application must
 * define EMBER_APPLICATION_USES_SOURCE_ROUTING in its
 * configuration header to use this.
 */
void emberIncomingRouteRecordHandler(EmberNodeId source,
                                     int8u relayCount,
```

```
                                        EmberMessageBuffer header
                                        int8u relayListIndex);

/** @description The application can implement this callback to
 * supply source routes to outgoing messages. The application
 * must define EMBER_APPLICATION_USES_SOURCE_ROUTING in its
 * configuration header to use this. It uses the supplied
 * destination to look up a source route. If available, it
 * appends the source route to the supplied header using the
 * proper frame format, as described in section 3.4.1.9
 * "Source Route Subframe Field" of the ZigBee specification.
 *
 * @param destination: The network destination of the message.
 * @param header: The message buffer containing the partially
 * complete packet header. The application appends the source
 * route frame to this header.
 */
void emberAppendSourceRoute(EmberNodeId destination,
                            EmberMessageBuffer header);
```

The first callback supplies the recorded routes which can be stored in a table. The second callback is invoked by the network layer for every outgoing unicast (including APS acknowledgements), and it is up to the application to supply a source route or not. The source route adds (#relays + 1) * 2 bytes to the network header frame, which therefore reduces the maximum application payload available for that packet.

The files *app/util/source-route.c* and *app/util/source-route.h* implements these callbacks and can be used as-is by node applications wishing to be a concentrator.

More information

The reader can find additional information in the online API reference guide and in the sample applications available on the EmberZNet Stack.

7.2 Link quality

7.2.1 Introduction

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[(b)] routers (ZR) keep track of inbound link quality in the neighbor table, typically by averaging LQI (Link Quality Indication) 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 via periodic one hop broadcasts, referred to as "link status" messages. The link status algorithm is explained below, as implemented in EmberZNet. It has been in

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

use in EmberNet since 2003 and in EmberZNet since 2005. It was formerly called the Neighbor Exchange mechanism.

Note that link status messages are handled automatically by the stack. Application writers need not be concerned with it. This information is provided for those wishing to understand the details of the network layer's operation, which can prove useful during troubleshooting.

7.2.2 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 via 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.

7.2.3 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 8 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.

Upon receipt of a link status message, either there is already a neighbor entry 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`.

7.2.4 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.

7.2.5 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.

7.2.6 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, patent-pending technology to manage the selection of neighbors in dense networks to optimize network connectivity.

7.3 Cluster library

7.3.1 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 data.

To ease 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 also define their own clusters, in the case that the pre-defined clusters do not meet their specific application needs.

ZigBee Application profiles will then reference which clusters are used within the profile, and will specify which clusters are supported by each device defined within the profile - some clusters will be 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.

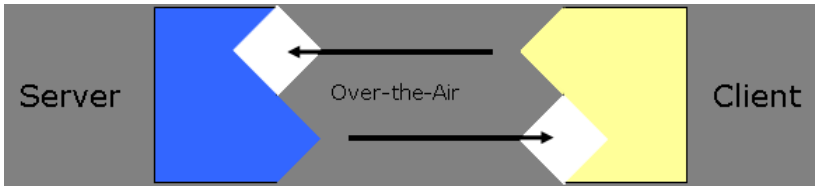
EmberZNet provides source code to easily assemble and disassemble ZCL messages, whether they are pre-defined by the ZCL or custom messages created by the developer.

7.3.2 ZigBee cluster library: inside clusters

Clients and servers

Each cluster is divided into two ends, described as a client end and a server end. The client end of a cluster will send messages that may be received by the server end, and vice/versa. Further, 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.

Figure 25. OTA Client - Server



This equality complicates discussions; for clarity, this document will always refer 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).

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 20* lists the most common data types. The data types are fully described in The ZCL Specification.[c] *Table 21* is an excerpt from that document.

c. See the ZCL Specification, 075123r02, Table 15.2.

Table 20. 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 21. 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	-	-	

Table 21. ZCL data types (continued)

Type class	Data type ID	Data type	Length of data (octets)	Invalid number	Analog / Discrete
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	
	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	-	-	

Table 21. ZCL data types (continued)

Type class	Data type ID	Data type	Length of data (octets)	Invalid number	Analog / Discrete
Identifier	0xE8	Cluster ID	2	0xFFFF	D
	0xE9	Attribute ID	2	0xFFFF	
	0xEA	BACnet OID	4	0xFFFF FFFF	
	0xEB - 0xEF	Reserved	-	-	
Miscellaneous	0xF0	IEEE address	8	0xFFFF FFFF FFFF FFFF	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.

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. *Table 22* lists profile-wide commands.

Table 22. 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 will specify which cluster and which attributes are to be accessed or modified. Additionally, each cluster will define which attributes support which commands - for example, an attributes may be declared READ ONLY, in which case it will not support the Write Attributes 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 are advised to review the ZCL Specification (075123r02).

Cluster specific commands

The payload format, support requirements (mandatory, optional), and behavior upon receipt of a cluster specific command are all defined in the cluster definition. Typically, these

commands will affect the state of the receiving device and may alter the attributes of the cluster as a side-effect.

For example, the ZCL General document[d] defines three commands that are received by the On/Off cluster server.[e] OFF, ON, and TOGGLE. 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). Section 12.3.4 of the ZCL General Document defines that the On/Off cluster client is responsible for generating the commands received by the server.

7.3.3 Walkthrough: Temperature measurement sensor cluster

To help solidify your understanding of these concepts, consider a portion of the Temperature Measurement Sensor cluster that is fully described in the ZCL: Measurement and Sensing document (053906).[f]

Table 23 is taken directly from the ZCL Measurement document mentioned in the previous paragraph.

Table 23. Temperature measurement sensor server attributes

Identifier	Names	Types	Range	Access	Default	Mandatory / Optional
0x0000	Measured Value	Signed 16-bit Integer	MinMeasured Value to MaxMeasured Value	Read only	0	M
0x0001	MinMeasured Value	Signed 16-bit Integer	0x954B - 0x7FFE	Read only	-	M
0x0002	MaxMeasured Value	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
As you can see, 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.

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 data information, either actively (through a read attributes command) or passively (through first a configure report attributes command and then from report attributes).

d. See the ZCL General document 053936r05.

e. See section 12.2.3 of the ZCL General Document 053936r05.

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

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 will support all basic commands. In the case of the MaxMeasuredValue, for example, only those to read and write attributes. Note: while the incoming write attributes command will be supported, in this case it will always generate 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 (and to test your understanding on a much more complicated cluster), see the Thermostat Cluster in the ZCL functional domain: HVAC document (06014).

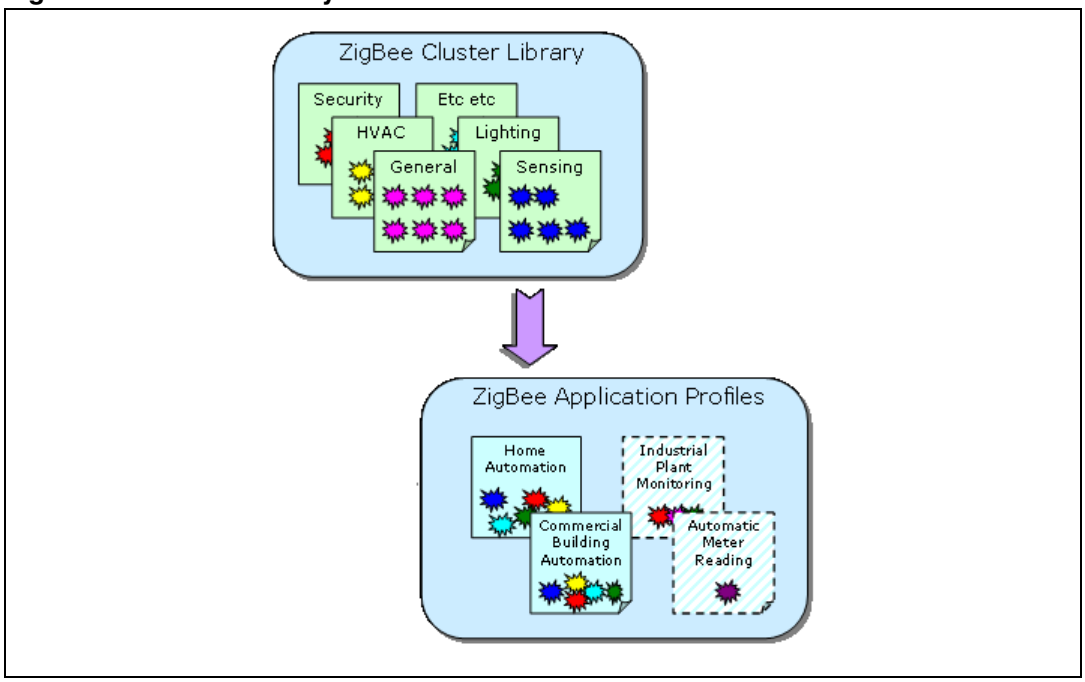
7.3.4 ZigBee cluster library: functional domains

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 26*).

Figure 26. Cluster library functional domains



ZigBee cluster library: 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.

7.4 Extended PAN IDs

For developers who have used earlier ZigBee stack software, the is 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.

`EmberNetworkParameters` struct now contains an 8 byte extended PAN ID in addition to the 2 byte PAN ID. Warning: your application must set this value, if only to zero it out. If it doesn't, it is likely to be initialized with random garbage which will lead to unexpected behavior.

`emberFormNetwork()` stores the extended PAN ID to the node data token. If a value of all zeroes is passed, a random value is used. In production applications it is strongly recommeded that the random XPID is used. Using a fixed value (such as the EUI64 of the coordinator) can easily lead to XPID conflicts if another network running the same application is nearby or if the coordinator is used to commission two different neighboring networks.

`emberJoinNetwork()` now joins to a network based on the supplied Extended PAN ID, rather than the PAN ID. For back-compatibility, if an Extended PAN ID of all zeroes is supplied, it will join based on the short PAN ID, and the Extended PAN ID of the network will be retrieved from the beacon and stored to the node data token.

Also, the API call for `emberJoinNetwork()` has changed. The `joinSecurely` parameter has been removed due to changes in the security model. Here is the new API function:

```
EmberStatus emberJoinNetwork(EmberNodeType nodeType,
EmberNetworkParameters *parameters)
```

There is a new API function to retrieve the Extended PAN ID:

```
void emberGetExtendedPanId(int8u *resultLocation);
```

7.5 ZigBee network rejoin strategies

End devices (ZED) that have lost contact with their parent or any node that does not have the current network key should call the API function shown below.

```
EmberStatus emberRejoinNetwork(boolean haveCurrentNetworkKey);
```

The haveCurrentNetworkKey variable determines if the stack performs a secure network rejoin (`haveCurrentNetworkKey` = TRUE) or an insecure network rejoin (`haveCurrentNetworkKey` = FALSE). The insecure network rejoin only works if using the

Commercial Security Library. In that case the current Network Key will be sent to the rejoining node encrypted at the APS Layer with that device's Link Key.

7.6 ZigBee messaging

Some key changes have occurred in how messaging is accomplished from previous versions of EmberZNet's ZigBee stacks.

The main changes are:

- There is now an address table, which is just an array of paired long and short addresses. This replaces the volatile (RAM) bindings that were used to target message destinations.
- Unicasts can be sent using either directly to a specific address, via an address table entry, or via a binding.
- Datagrams, sequenced messages, and the SPDO are no longer used in EmberZNet.

7.6.1 Cluster IDs

As noted earlier in this chapter, Cluster IDs are now 16 bits long instead of 8. All uses of cluster IDs in the API have been changed from `int8u` to `int16u`. In places where an array of cluster ID is used, such as in the `EmberEndpointDescription` type, the size field is the number of cluster IDs, not the number of bytes. Over the air cluster IDs are sent least-significant byte first, as is standard in ZigBee.

7.6.2 APS frame

The `EmberApsFrame` struct has changes that correspond to the changes in the actual APS frame:

- The cluster ID is an `int16u` instead of an `int8u`.
- There is one-byte sequence field that contains the sequence number. This is only valid for incoming messages. For outgoing messages the stack ignores the value supplied by the application.

Because the APS frame is used for multicasts and broadcasts as well as unicasts, the ZigBee standard options have changed from `EMBER_UNICAST_OPTION_`... to `EMBER_APS_OPTION_`.... See below for a complete list of the options. `EmberUnicastOption` has changed to `EmberApsOption` and changed size from `int8u` to `int16u`.

There is a two-byte `groupID` field. This is valid only for incoming multicasts and broadcasts. For broadcasts the field will contain the destination address.

The following are standard ZigBee APS options which can be used when sending packets and be read on the receiver.

```
EMBER_APS_OPTION_RETRY
EMBER_APS_OPTION_SECURITY (includes a new Encrypt message using APS-
level security.)
EMBER_APS_OPTION_SOURCE_EUI64 (includes a new Include for the source
EUI64 in the network frame.)
EMBER_APS_OPTION_DESTINATION_EUI64 (now includes the destination
EUI64 in the network frame.)
```

The following are EmberZNet options that control message transmission. These are not available on the receiver.

```
EMBER_APS_OPTION_ENABLE_ROUTE_DISCOVERY
EMBER_APS_OPTION_FORCE_ROUTE_DISOCVERY
EMBER_APS_OPTION_ENABLE_ADDRESS_DISCOVERY (new)
EMBER_APS_OPTION_POLL_RESPONSE
```

ZigBee has removed indirect APS messages. The following options will no longer be available:

```
EMBER_UNICAST_OPTION_APS_INDIRECT
EMBER_UNICAST_OPTION_HAVE_SOURCE
```

7.6.3 Address table

EUI64 values to network ID mappings will be kept in an address table. The stack will update the node IDs as new information arrives. It will not change the EUI64s.

```
EmberStatus emberSetAddressTableRemoteEui64(int8u
addressTableIndex, EmberEui64 eui64);
void emberSetAddressTableRemoteNodeId(int8u addressTableIndex,
EmberNodeId id);


void emberGetAddressTableRemoteEui64(int8u addressTableIndex,
EmberEui64 eui64);
EmberNodeId emberGetAddressTableRemoteNodeId(int8u
addressTableIndex);
```

The size of the table can be set by defining `EMBER_ADDRESS_TABLE_SIZE` before including ember-configuration.c.

The binding table will have its own remote ID table, just as in EmberZNet 2.x. Function changes include:

```
emberGetBindingDestinationNodeId() has been renamed to
emberGetBindingRemoteNodeId()
emberSetBindingDestinationNodeId() has been renamed to
emberGetBindingRemoteNodeId()
```

These functions perform the same function as they did in EmberZNet 2.x, but the names have been changed to to be consistent with the address calls. There are 4 reserved node ID values that are used with the address and binding tables. These are described in *Table 24*.

Table 24. Bindings remote ID table functions

Function	Description
`EMBER_TABLE_ENTRY_UNUSED_NODE_ID (0xFFFF)`	This value is used when setting or getting the remote node ID in the address table or getting the remote node ID from the binding table. It indicates that address or binding table entry is not in use.
`EMBER_MULTICAST_NODE_ID (0xFFFE)`	This value is returned when getting the remote node ID from the binding table and the given binding table index refers to a mulicast binding entry.

Table 24. Bindings remote ID table functions

Function	Description
`EMBER_UNKNOWN_NODE_ID (0xFFFD)`	This value is used when getting the remote node ID from the address or binding tables. It indicates that the address or binding table entry is currently in use but the node ID corresponding to the EUI64 in the table is currently unknown.
`EMBER_DISCOVERY_ACTIVE_NODE_ID (0xFFFC)`	This value is used when getting the remote node ID from the address or binding tables. It indicates that the address or binding table entry is currently in use and network address discovery is underway.

There is a new function that will validate that a given node ID is valid and not one of the reserved values.

```
boolean emberIsNodeIdValid(EmberNodeId id);
```

There are also two new functions that search through all the relevant stack tables (address, binding, neighbor, child) to map a long address to a short address, or vice versa.

```
EmberNodeId emberLookupNodeIdByEui64(EmberEUI64 eui64);
EmberStatus emberLookupEui64ByNodeId(EmberNodeId nodeId,
EmberEUI64 eui64Return);
```

Since end device children can no longer be identified by their short ID, there are two new functions to allow the application to set and read the flag that increases the interval between APS retry attempts.

```
boolean emberGetExtendedTimeout(EmberEUI64 remoteEui64);
void emberSetExtendedTimeout(EmberEUI64 remoteEui64, boolean
extendedTimeout);
```

7.6.4 Sending messages

The destination address for a unicast can be obtained from the address or binding tables, or passed as an argument. There is an enumeration for indicating which is to be used for a particular message. The same enumeration is used with `emberMessageSent();` `EMBER_OUTGOING_BROADCAST` and `EMBER_OUTGOING_MULTICAST` cannot be passed to `emberSendUnicast()`.

Use of the address or binding table allows the stack to perform address discovery by setting the EMBER_APS_OPTION_ENABLE_ADDRESS_DISCOVERY option.

```
enum {
  EMBER_OUTGOING_DIRECT,
  EMBER_OUTGOING_VIA_ADDRESS_TABLE,
  EMBER_OUTGOING_VIA_BINDING,
  EMBER_OUTGOING_BROADCAST,    // only for emberMessageSent()
  EMBER_OUTGOING_MULTICAST     // only for emberMessageSent()
};
typedef int8u EmberOutgoingMessageType;
EmberStatus emberSendUnicast(EmberOutgoingMessageType type,
                             int16u indexOrDestination,
                             EmberApsFrame *apsFrame,
                             EmberMessageBuffer message);
```

EMBER_OUTGOING_VIA_BINDING uses only the binding's address information. The rest of the binding's information (cluster ID, endpoints, profile ID) can be retreived using emberGetBinding() and emberGetEndpointDescription().

The next snippet of example code shows how a message might be sent using a binding. Using the options in the example duplicates the behavior of emberSendDatagram() (for a non-aggregation binding; when sending to an aggregator route and address discovery should not be enabled).

```
EmberApsFrame apsFrame = {
  profileId,
  clusterId,
  sourceEndpoint,
  destinationEndpoint,
  EMBER_APS_OPTION_RETRY
    | EMBER_APS_OPTION_ENABLE_ROUTE_DISCOVERY
    | EMBER_APS_OPTION_ENABLE_ADDRESS_DISCOVERY
    | EMBER_APS_OPTION_DESTINATION_EUI64
};
emberSendUnicast(EMBER_OUTGOING_VIA_BINDING,
                 bindingIndex,
                 &apsFrame,
                 message);
```

emberSendReply() is unchanged. It can be used to send a reply to any retried APS messages. Replies are a nonstandard extension to ZigBee.

```
EmberStatus emberSendReply(int16u clusterId, EmberMessageBuffer
reply);
```

Multicasts get an APS frame plus radii. The groupId is specified in the APS frame. The nonmemberRadius specifies how many hops the message should be forwarded by devices that are not members of the group. A value of 7 or greater is treated as infinite. There is no longer a separate limited multicast API call.

```
EmberStatus emberSendMulticast(EmberApsFrame *apsFrame,
                               int8u radius,
                               int8u nonMemberRadius,
                               EmberMessageBuffer message);
```

There is a new multicast table. The size is EMBER_MULTICAST_TABLE_SIZE and defaults to 8. Multicast table entries should be created and modified manually by the application; just index into the array.

```
  EmberMulticastTableEntry *emberMulticastTable;
/** @brief Defines an entry in the multicast table.
   * @description A multicast table entry indicates that a
   * particular endpoint is a member of a particular multicast
   * group. Only deviceswith an endpoint in a multicast group will
   * receive messages sent to that multicast group.
   */
  typedef struct {
    /** The multicast group ID. */
    EmberMulticastId multicastId;
    /** The endpoint that is a member, or 0 if this entry is not in
     * use (the ZDO is not a member of any multicast groups).
     */
    int8u  endpoint;
```

```
} EmberMulticastTableEntry;
```

Now that ZigBee has three different broadcast addresses (everyone, rx-on-when-idle only, routers only), broadcasting requires specifying a destination address. On the receiver, this can be read out of the groupId field in the apsFrame.

```
#define EMBER_BROADCAST_ADDRESS 0xFFFC
#define EMBER_RX_ON_WHEN_IDLE_BROADCAST_ADDRESS 0xFFFD
#define EMBER_SLEEPY_BROADCAST_ADDRESS 0xFFFF

EmberStatus emberSendBroadcast(EmberNodeId destination,
                               EmberApsFrame *apsFrame,
                               int8u radius,
                               EmberMessageBuffer message);
```

7.6.5 Message status

`emberMessageSent()` should be called for all outoing messages. The type of message is given by the enumeration of outgoing message types. `emberMessageSent()` will be called:

- For retried unicasts, when an ACK arrives or the message times out.
- For non-retried unicasts, when the MAC receives an ACK from the next hop or the MAC retries are exhausted.

For broadcasts and multicasts, when the message is removed from the retry queue (not the broadcast table).

The `indexOrAddress` argument is the destination address for direct unicasts, broadcasts, and multicasts. For unicasts sent via the address or binding tables it is the index into the relevant table. *Table 25* summarizes the status arguments for broadcasts and multicasts.

Table 25. Status argument for broadcasts and multicasts

Status Argument	Description
`EMBER_DELIVERY_FAILED`	If the message was never transmitted.
`EMBER_DELIVERY_FAILED`	If radius is greater than 1 and we don't hear at least one neighbor relay the message.
`EMBER_SUCCESS`	Otherwise.

Example code:

```
void emberMessageSent(EmberOutgoingMessageType type,
                      int16u indexOrDestination,
                      EmberApsFrame *apsFrame,
                      EmberMessageBuffer message,
                      EmberStatus status);
```

7.6.6 Disable relay

If `EMBER_DISABLE_RELAY` is defined in the app configuration header, the node will not relay unicasts, route requests, or route replies. It can still generate route requests and replies, so that it can be the source or destination of network messages. This is intended for use in a gateway.

7.6.7 Incoming messages

`emberIncomingMessageHandler()` is unchanged from EmberZNet 2.x. The possible message types are:

- `EMBER_INCOMING_UNICAST`
- `EMBER_INCOMING_UNICAST_REPLY`
- `EMBER_INCOMING_BROADCAST`
- `EMBER_INCOMING_BROADCAST_LOOPBACK`
- `EMBER_INCOMING_MULTICAST`
- `EMBER_INCOMING_MULTICAST_LOOPBACK`

For incoming broadcasts, the destination broadcast ID will be in the groupId field of the APS struct.

`EMBER_INCOMING_SHARED_MULTICAST`, which indicated that there were multiple bindings whose group matched an incoming multicast in EmberZNet 2.x, is no longer being used. Instead, the procedure shown below should be used to walk through the matching bindings. This code snippet examines the bindings starting from startIndex and returns the index of the first multicast binding for the specified group. 0xFF is returned if there are no matching bindings.

```
int8u emberNextMatchingMulticastBinding(int16u groupId,
  int8u startIndex,
  EmberBindingTableEntry *binding);
```

There is one additional function that can only be called from within `emberIncomingMessageHandler()`. This extracts the source EUI64 from the network frame of the message, but only if `EMBER_APS_OPTION_SOURCE_EUI64` is set.

```
EmberStatus emberGetSenderEui64(EmberEUI64 senderEui64);
```

7.6.8 Binding

The binding table is now used only in support of provisioning. The idea is that a provisioning tool will use the ZDO to discovery the services available on a device and to add entries to the device's binding table. Other than the ZDO, no use is made of the binding table within ZigBee. It is up to the application to make use of the information in the table how it sees fit.

Messages now use an Address Table to establish the message destination. This has had the side benefit of allowing us to put the binding table into a library, freeing up flash for those applications that do not need it.

8 Bootloading

This release includes full support for the Serial-UARTstandalone bootloader and provide reference examples for implementing application bootloaders on the STM32W108 platform (providing stub library APIs for EEPROM). Although operation of the bootloaders for the STM32W108 is very similar to that on the SN2xx platforms, the content of this chapter has been only partially updated with the differences on the STM32W108 platform.

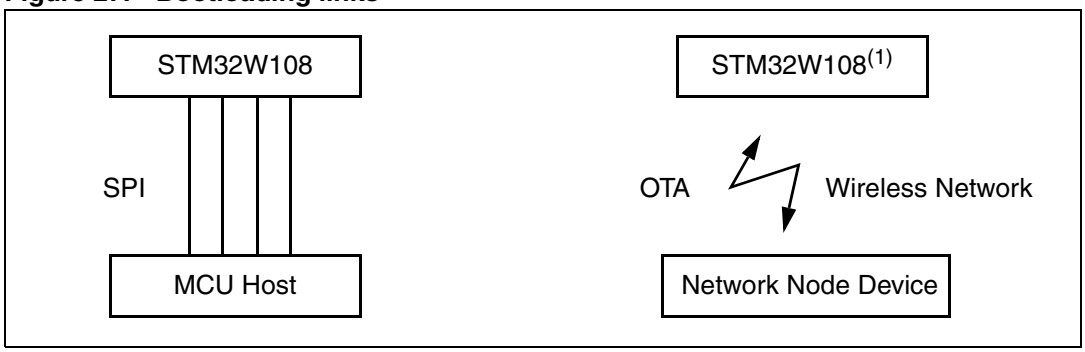
In the next release, this chapter will be updated to fully document the STM32W108 platform.

8.1 Introduction

The bootloader is a program stored in reserved flash memory that allows the node to update its image on demand, either via serial communication or over the air. Production level programming is typically accomplished 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 after deployment with new features and bug fixes. The bootloading capability of these Networking Devices 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 27*.

Figure 27. Bootloading links



1. To be supported in a future release for OTA portion only.

Two types of bootloaders are available with EmberZNet's ZigBee Networking Devices: Standalone and Application. These two bootloaders differ in the amount of flash required and location of the stored image as discussed below.

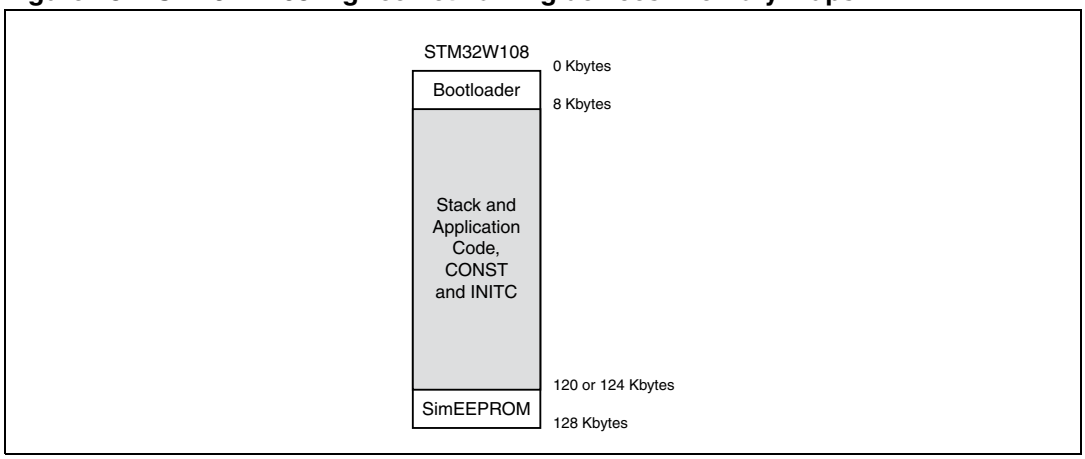
Devices without a bootloader have no supported way of upgrading the firmware over the air once deployed. It is for this reason that EmberZNet advocates implementing a bootloader.

In all the bootloading situations described in this document, it is assumed that the source node acquires the new firmware version to be bootloaded through some other means. This necessary part of the bootloading process is system-dependent and beyond the scope of this document.

8.1.1 Memory space for bootloading

Figure 28 shows the memory maps for the STM32W108 ZigBee Networking Device.

Figure 28. STM32W108 ZigBee networking devices' memory maps

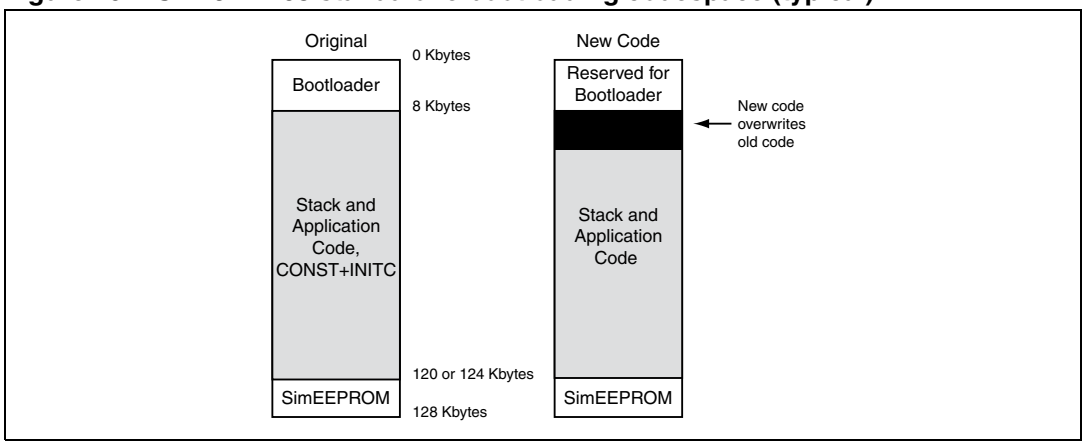


For the STM32W108, a block of 8 Kbytes of low (Flash) memory is reserved to hold the bootloader, as well as either 4 Kbytes or 8 Kbytes of high (Flash) memory for the simulated EEPROM. In all cases, the balance of the memory space is unreserved and available to hold the stack and application code.

8.1.2 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. There is very little interaction between the standalone bootloader and the application running in Flash memory. In general, the only time that the application interacts with the bootloader is when it wants to run the bootloader in which it will call `halLaunchStandaloneBootloader()`. Once the bootloader is running, it receives bootload packets containing (new) firmware image either via physical connections (for example, serial, SPI) or via the radio (over-the-air). *Figure 29* illustrates what happens to the device's Flash memory during bootloading.

Figure 29. STM32W108 standalone bootloading codespace (typical)

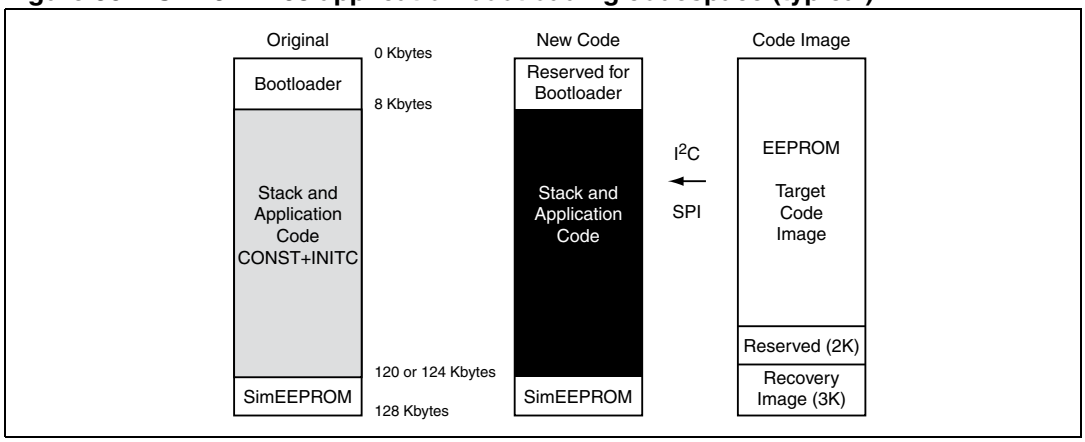


When bootloading is initiated, the new code is overwritten onto the existing Stack & Application Code. If any errors occur during this process, the code cannot be recovered and bootloading must start over at byte zero.

8.1.3 Application bootloading

The application bootloader relies on the application to perform the upgrade process. There are three components to application bootloading: the application, the application bootloader, and the recovery image. There is also an area 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. 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 EEPROM 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 30* shows a typical memory map for the application bootloader.

Figure 30. STM32W108 application bootloading codespace (typical)



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

8.2 Design decisions

Table 26 lists some major considerations for each type of bootloading. These are the trade-offs that must be considered by the systems designer.

Table 26. Design trade-offs

Standalone bootloading	Application bootloading
Self Contained	Additional Hardware & Application Code Required
Serial Link	Serial Link
OTA Link (restricted to single hop communication)	OTA Link (multi hop capable)

Table 26. Design trade-offs

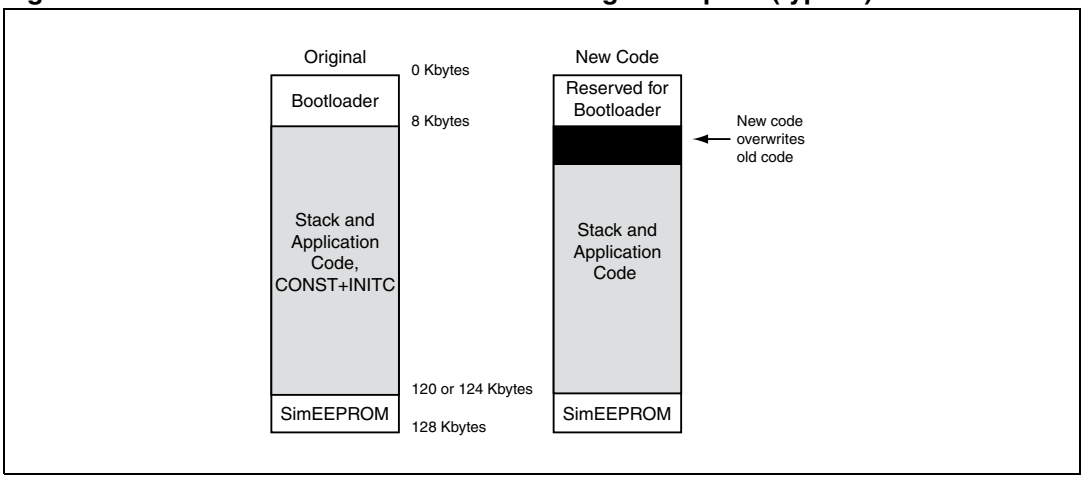
Standalone bootloading	Application bootloading
Multiple Modes (Clone, Recovery, Pass-Through, Serial)	Fewer Modes (Clone & Serial)
Larger memory requirement for bootloader code on some platforms	Smaller memory requirement for bootloader code on some platforms
Reduced code space for Stack & Application Code	Maximum code space for Stack & Application Code
Bootload error may require retransmission of code from external source	Bootload errors can be repaired from saved code image

8.3 Standalone bootloading

8.3.1 Introduction

A standalone bootloader is a program that operates both the serial port and the radio in a single-hop, MAC-only mode. 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 31* depicts a sample memory map and how the bootloading process over-writes 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 can reside in this code space. A failure during bootloading means that the process must begin again.

Figure 31. STM32W108 standalone bootloading codespace (typical)



When bootloading is initiated, the new code is overwritten onto 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.

8.3.2 Serial and OTA modes

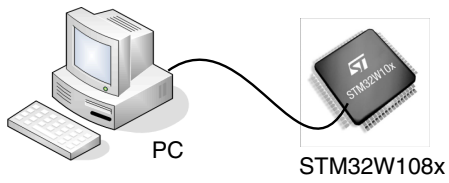
The standalone bootloader and its utility library support three basic modes for uploading an application image to a network device:

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

8.3.3 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.

Figure 32. Serial upload



A serial connection is established as follows:

1. The PC connects to the target device at 115,200 baud (38,400 for the AVR128), 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:

```
STM32W108 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 this 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.

Serial upload: uploading an image

Selection of the menu option **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 image 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.

Serial upload: errors

If an error occurs during the upload, the bootloader displays one of the errors shown in *Table 27*, then displays the message Serial upload aborted and returns to the bootloader menu.

Table 27. Serial uploading error messages STM32W108

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.
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.

Running the application image

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

Obtaining image information

On the STM32W108 platform, the image info is customizable by the user, and can be specified as a string using the `--imageinfo` option in the EmberZNet `em3xx_convert` utility, which creates the ebl image from an s37. Menu option 3 then displays the information as a quoted string, similar to the following:

```
"custom image info"
```

8.3.4 Over-the-air upload

Note: The STM32W108 Platform does not yet support over-the-air uploads with the Standalone Bootloader. The following sections show how the over-the-air uploads will work when supported by the STM32W108 platform.

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

- Passthrough
- Multi-hop Passthrough
- Cloning

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, which 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.

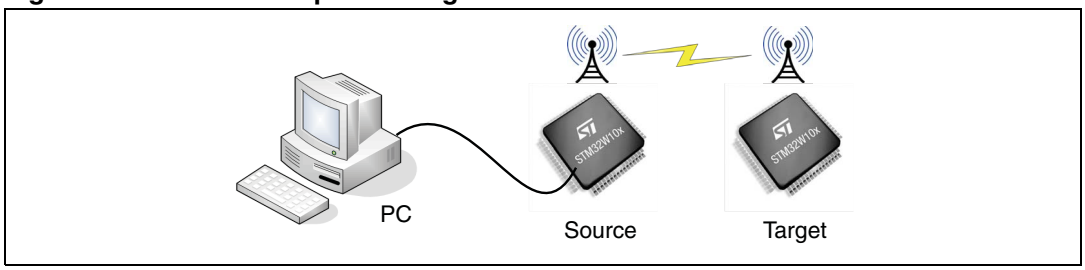
During over-the-air upload, only the target device actually runs the bootloader. The source device and any intermediary devices that participate in the upload process continue to run an application that is based on the EmberZNet stack.

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.

Passthrough

When using over-the-air passthrough mode, the source node is connected to a PC via a serial cable. The source receives an image over the serial line and passes the image to the target node over the air.

Figure 33. 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 EmberZNet standalone bootloader uses for its over-the-air protocol.

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`.

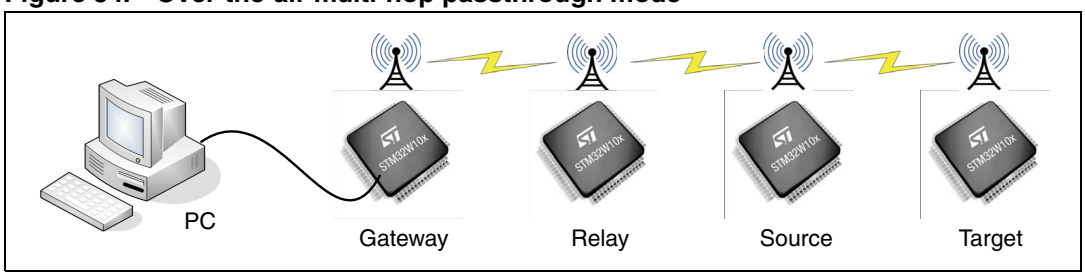
After receiving the query response from the target node, the source node calls `emberIncomingBootloadMessageHandler()`, which directs the serial download by calling `XModemReceiveAndForward()`.

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

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. 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.

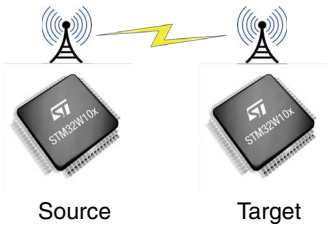
Figure 34. Over-the-air multi-hop passthrough mode



Cloning

A device can clone its own application image and upload it to the target device:

Figure 35. Over-the-air cloning mode



Cloning requires the source device to convert its application image in flash to an .ebl image that it can send to the target device.

Cloning: Target Node Requirements

In order to upload a cloned application, the target node must be running in the bootloader. The node can be in bootloader mode for two reasons:

- The application launched bootload mode by calling `halLaunchStandaloneBootloader()`.
- The node is in recovery mode because it has an invalid or corrupt image. For more on recovering a node, see *Section 8.3.6: Upload recovery*.

8.3.5 Hybrid mode uploads

Upload operations can mix any of the modes previously described. For example, you can combine cloning with multi-hop passthrough, or 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.

Note: OTA can be target or source.

8.3.6 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. In this case, the bootloader restarts and continues to listen on the same channel for any retries by the source. 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 via the hardware trigger, the target enters bootload recovery mode. In this mode, the bootloader listens on the default channel (13) for a new upload to begin.

The STM32W108 uses PA5 as a hardware-based trigger for recovery mode. Holding this pin low during power-up or across a reset causes the STM32W108 to enter a special ROM-based bootloader. Sending a carriage return at 115200 baud then causes the standalone bootloader to launch. The STM32W108 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. An example of utilizing PA7, which is connected to a button on the STM32W108, is provided.

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()`. Prior to calling the function, the source node needs to ensure that it is on the same channel as the node to be recovered. The

source node can leave the current channel and join or form the network on 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.
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.

8.3.7 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: It is recommended that you do not modify the supplied utilities.

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

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

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`.

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);
```

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))
```

Application requirements

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

- Include bootload-utils.h in the application's .h file.
- Define (in preprocessor definitions for the build) the application's `APPLICATION_TOKEN_HEADER` file as *bootload-utils-token.h*, or another header file that includes *bootload-utils-token.h*.
- Include the *bootload-utils.c* file in the project.
- Define `EMBER_APPLICATION_HAS_BOOTLOAD_HANDLERS` and #define `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, the customer can reduce flash requirements by eliminating features not needed. Typical uses are to remove bootloader support (if not supporting bootloading of legacy STMicroelectronics devices) or to remove passthrough or cloning support.

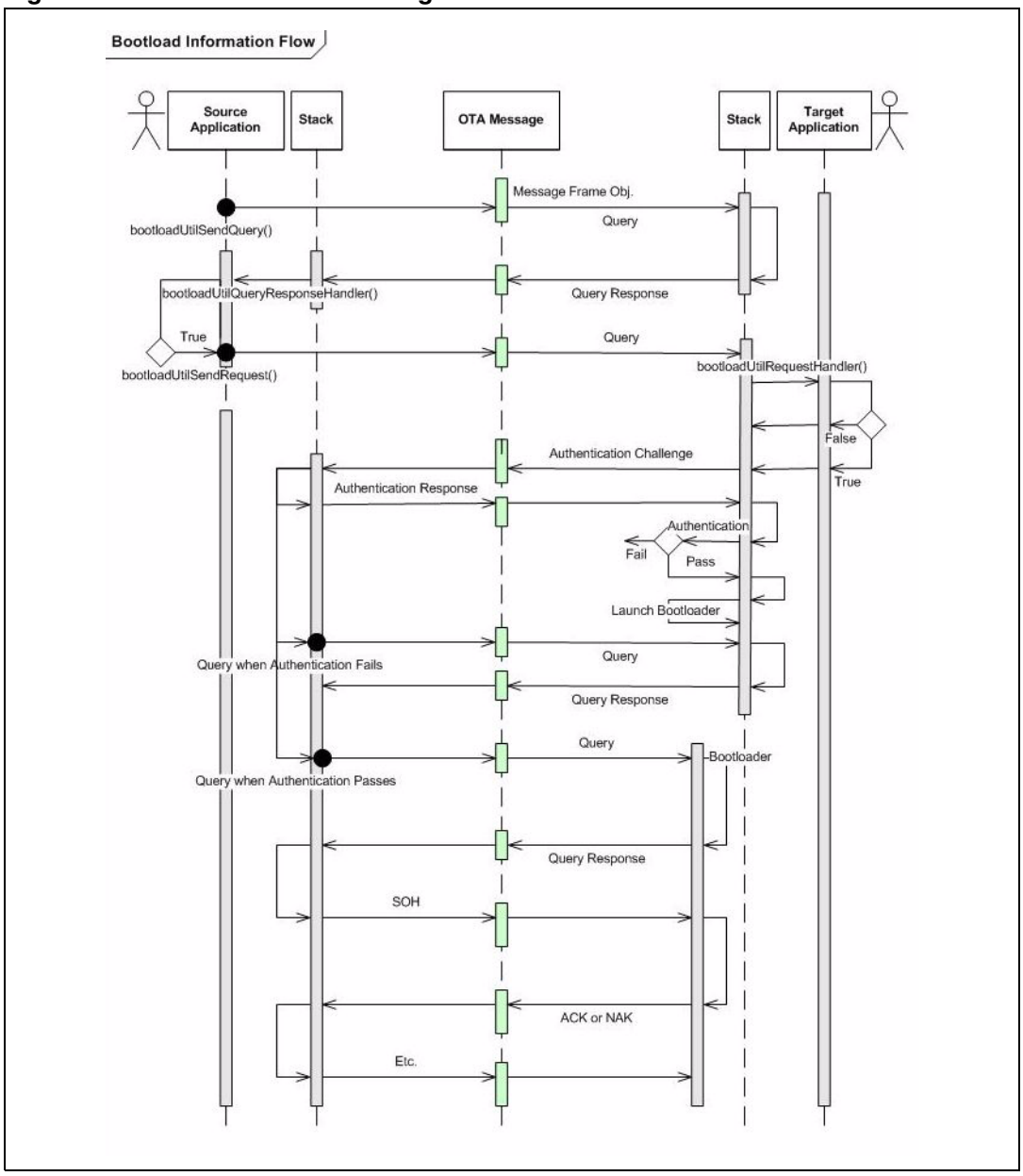
Modify the following definitions in bootloader-demo-v2-configuration.h:

```
USE_BOOTLOADER_LIB          : always defined with the bootloader
demo library
SBL_LIB_SRC_NO_CLONE       : define this to build a node without
cloning ability
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_LEGACY_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.
```

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 process is summarized in *Figure 36*.

Figure 36. 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 in order to verify that the source node is authorized to initiate the request.
   - Desired mode: Set to `BOOTLOAD_MODE_PASSTHRU` or `BOOTLOAD_MODE_CLONE`.
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.

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.

8.3.8 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. EmberZNet provides em3xx_load.exe utility to set these tokens.

See application note *SettingManufacturingCertificates.doc* for the STM32W108 and either of the Bringing Up Custom Nodes for the STM32W108 Application Notes 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 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. STMicrolectronics 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.

8.3.9 Example standalone bootloading scenario

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.

Serial baud rates and ports used

The original sample application requires a serial port to communicate with users. All commands are transmitted and received via serial port 1. Another serial port (0) is required to be configured as the bootload port at 115200 bps (it is used for OTA mode not yet supported with STM32W108). User is requested to adapt the sample application for matching its platforms requirements.

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 28. Serial commands supported in the full-featured sample application

Command	Description
clone target-EUI64	Upload a copy of the node's application image to the specified target node.
default mode	Recover nodes that fail bootloading on the default channel (13), where mode is set to 1 (passthrough), 2 (clone), or 3 (V1 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), 2 (clone), or 3 (v1 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.

Note: Currently the sample application for the STM32W108 supports only these commands: Form, Join, Leave and Serial.

Usage notes

To start bootloading, select the .ebl file to download using a terminal emulator program (such as Hyperterm) connected to the node's serial port. The node signals the event by printing a stream of C characters to serial port 1.

8.3.10 V2 standalone bootloader protocol

The following sections describe the purpose of each packet and when it is used.

Table 29. 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

Table 29. Broadcast query message format (continued)

# bytes	Field	Description/notes
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	

Common header used for all other message types

Table 30 describes the common header that is used for all other message types.

Table 30. 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

8.3.11 Other packets

Query packet

Table 31. Query packet

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

Query response

Table 32. 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
1	Micro	0x01
1	PHY	0x02
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	

Bootloader launch request

Table 33. 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	

Bootloader authorization challenge

Table 34. Bootloader authorization challenge

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

Bootloader authorization response

Table 35. Bootloader authorization response

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

XModem SOH

Table 36. 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	

XModem EOT

Table 37. XModem EOT

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

XModem ACK

Table 38. XModem ACK

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

XModem NACK

Table 39. XModem NACK

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

XModem Cancel

Table 40. XModem Cancel

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

XModem Ready

Table 41. XModem Ready

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

8.4 Application bootloading

8.4.1 Introduction

The application bootloader has the single purpose of reprogramming the flash with an application image stored in external memory, typically EEPROM. The application code performs the actual download process. The recovery image uses the serial port to load a new image when the application cannot; possibly due to a faulty or missing application. Because of this, the application bootloader is smaller and simpler than the standalone bootloader.

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 in external storage, the application doesn't need to be overwritten until the new image has been successfully saved. It also allows the possibility of a node saving an image in its EEPROM and forwarding that image to other nodes. . If extra EEPROM 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 to store the image in.

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

Definitions:

- Application Code (or just application) - software that provides the nodes' primary functionality
- Application Bootloader - a bootloader that relies on the application software to upload a new primary image
- Recovery Image - software that allows a device with a corrupt application to be reloaded with a new image
- Source node - the source of the upload image

  Target node - the recipient of the upload image

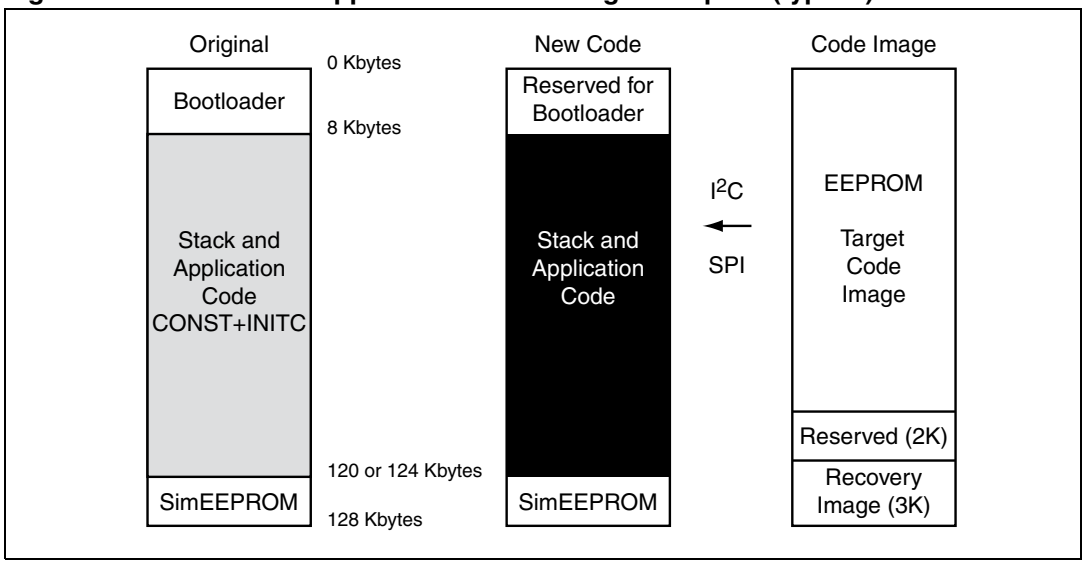
8.4.2 Memory map

The application bootloader uses the same memory map as the standalone bootloader with two exceptions. The bootloader segment is smaller, 5 Kbytes versus 10 Kbytes, and there is an external EEPROM, which is addressed separately.

The STM32W108 uses an 8-Kbyte memmap for the application bootloader with an external EEPROM, which is addressed separately.

The application bootloader can access its remote storage using either an I2C or SPI interface. The application bootloader utilizes 256 bytes per page for a total of 128 Kbytes, the same size as the STM32W108 Flash. The recovery image is stored in the top 3 Kbytes of the EEPROM. A diagram of the memory layout is shown in *Figure 37*.

Figure 37. STM32W108 application bootloading codespace (typical)



The EmberZNet for STM32W108 release includes sample stubs drivers (I^2C, SPI) to be customized for accessing to customer SPI or I^2C EEPROM.

8.4.3 Modes

Using the application bootloader library provides over-the-air passthrough and serial image uploads.

8.4.4 Emergency recovery mode

The Emergency Recovery mode is used to recover the device when its normal application is unresponsive. The device can be forced into recovery mode and reprogrammed via serial connection. The STM32W108 uses PA5 as a hardware-based trigger for recovery mode. The main primary use of this pin is PTI_DATA. Holding this pin low during power up or across a reset causes the STM32W108 to enter a special ROM-based bootloader. Sending a carriage return at 115200 baud then causes the recovery mode of the application bootloader to launch. The STM32W108 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 PA7, which is connected to a button on the STM32W108, is provided.

Image corruption

It should be noted that even with a properly encoded .ebl file, a download operation can result in an unresponsive device. A properly encoded download file does not indicate the image's integrity. Logic errors could prevent the device from communicating over the air. In this situation the user would have to gain physical access to the device and perform a serial recovery operation. This scenario should only occur during development and underscores the importance of thoroughly testing any production download images.

Image version format

Bootloader image version is encoded in one 16bit integer in the following format. The high order byte contains the version info, the low order byte contains the build number. The high nibble of the high order byte contains the major version, the low nibble contains the minor

version. As an example, the version value 0x2107 would correspond to major version 2, minor version 1 and build number 07. The application bootloader version can be obtained in the app by calling `halAppBootloaderGetVersion()`.

```
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
x  x  x  x                                    major version (0 - 15)
         y  y  y  y                           minor version (0 - 15)
                  b  b  b  b  b  b  b  b  build number  (0 - 255)
```

8.4.5 Remote EEPROM connection

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

The application bootloader can access its remote storage using either an I^2C or SPI interface. It utilizes 256 bytes per page for a total of 128 Kbytes.

Both the I^2C and SPI sample versions of the application bootloader are supplied as reference to be customized by customers.

Remote EEPROM Access

The file bootloader-interface-app.c contains access routines for the EEPROM. The following routines are available:

`halAppBootloaderImageIsValidReset`

`halAppBootloaderImageIsValid`

`halAppBootloaderInstallNewImage`

`halAppBootloaderReadDownloadSpace`

`halAppBootloaderWriteDownloadSpace`

`halAppBootloaderGetImageData`

`halAppBootloaderGetVersion`

`halAppBootloaderGetRecoveryVersion`

`halAppBootloader`GetAppImageData

8.4.6 Loading

Application Bootloader

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

```
em3xx_load.exe app-bootloader.s37
```

Recovery image

The STM32W108 application bootloader contains the recovery image so there is no need to store it in the EEPROM.

8.4.7 Modes

The application bootloader relies on the application code, the application bootloader library and the recovery image to obtain new images. Using those components the following upload modes can be supported:

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

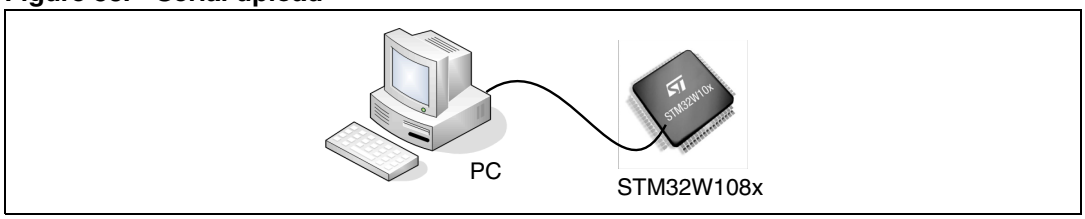
- Serial
- Passthrough
- Multi-hop passthrough

Serial upload

The recovery image provides the ability to perform serial uploads. The device is connected to a PC using a serial cable and the PC transfers the image to the device using the XModem protocol. The details of this mode are discussed in *Section 8.4.8: Recovery image*.

The application code can also perform serial upload using the bootloader library. The node connects to a PC using a serial cable. It uploads the image from the PC and stores it in the EEPROM. The bootloader can be invoked to flash the image. The image can also be forwarded to a remote node over-the-air.

Figure 38. Serial upload



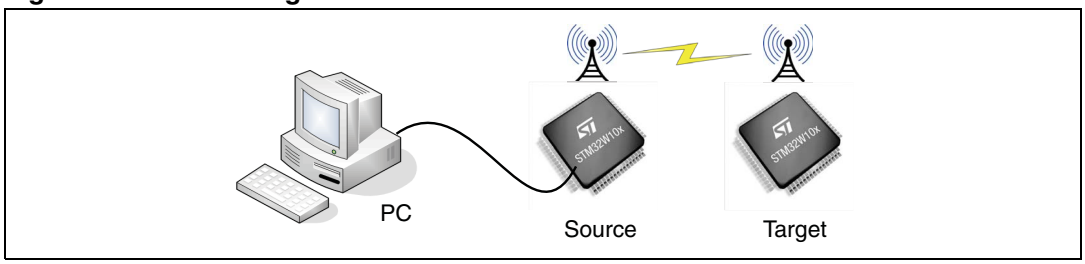
Over-the-air upload with application bootloader library

During over-the-air upload the source and target nodes are running their application code. The source transfers the image data in 64 byte, XModem formatted Zigbee protocol packets.

Passthrough

When using passthrough mode, the source node is connected to a PC via a serial cable. The source receives an image over the serial line and stores the image in EEPROM. It then passes the image to the target node over the air.

Figure 39. Passthrough mode

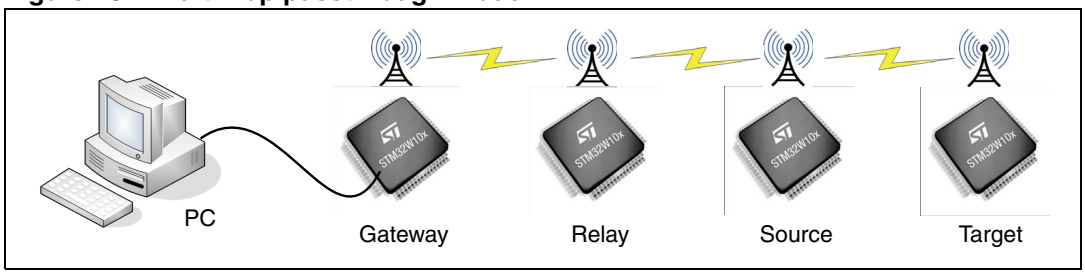


The PC is expected to transfer the .ebl image to the source node using standard 128-byte XModem CRC packets. The source node application must split these packets into two 64-byte chunks to be forwarded to the remote target using Zigbee protocol messages. The application code on the target node accepts the packets and stores the raw data in the EEPROM. Once the full image has been saved by the target node it can call `halAppBootloaderInstallNewImage()` to reset into the application bootloader and flash the new image.

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. 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.

Figure 40. Multi-hop passthrough mode



8.4.8 Recovery image

The recovery image is used as a failsafe mechanism to recover a module without a valid application image

On the STM3232W108 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.

Running the recovery image

The recovery image is only run in three instances. The first is at manufacturing time when the em3xx_load program initially loads it. The second is when the application bootloader invokes it. The third is when Emergency Recovery mode is initiated.

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. As previously mentioned, it first verifies that it has been copied to EEPROM. After the first invocation this test will succeed and the copy operation will not be performed.

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 115200 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 via serial cable.

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

Output of recovery image

There is no command line interface associated with the recovery image. Its only output is on the serial line connected to the terminal emulator. When it initially writes itself to EEPROM it emits a 'W' character for every 128 byte page it writes. After successfully writing the image

to EEPROM or verifying that the image has previously been saved to EEPROM, it emits 'OK'. Once the image is verified, it emits 'C's every second waiting for upload.

8.4.9 Errors during application bootloading

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 EEPROM 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 via `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 to EEPROM. A printable error string can be acquired from calling `halGetResetString()`. The application bootloader does not print out on the serial line.

Recovery image errors

If the recovery image encounters an error while uploading an image, it will print "Err" or "Stat" on the serial line followed by the error or status number in hex. It will then restart the upload process, emitting 'C' characters while waiting for a new upload to start.

Table 42. 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 EEPROM does not match data written
84	Image Size error: download image is greater than EEPROM space available

8.4.10 Application bootload libraries

The application bootloader utility library files (app-bootload-utils.c, app-bootload-utils.h, app-bootload-utils-internal.h, app-bootload-utils-configuration.h), in /app/util/bootload, provide APIs that source and target node applications can use to interact with an application bootloader. The library is supplied as source code.

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

The following sections discuss programming requirements for using the application bootloader.

- Library Interfaces
- Application Requirements
- Application Bootloading Process
- Library Features and Limitations
- Manufacturing Tokens

Library interfaces

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

Functions

```
// *****************************************************************
// Literals that are needed by the application.

/**@description The cluster id for application bootload. This defines a set
of commands/messages used by this particular application.
*/
#define APP_BOOTLOAD_CLUSTER_ID 0xE002

/**@description Selected endpoint used for application bootloading. This is
used to filtered incoming radio messages.
*/
#define APP_BOOTLOAD_ENDPOINT 240

/**@description A value that application can check whether bootloading is
// in progress. This is necessary because we want the application to be
// aware that bootloading is going on and it needs to limit its activities.
// For example, do not print anything to serial port when passthru
// bootloading is going on because it may violate XModem protocol, and try
// to limit radio activities to minimum to avoid any interruptions to
// bootload progress. Used in bootload state machine.
*/
enum {
BOOTLOAD_STATE_NORMAL,                  // initial state (no bootloading)
BOOTLOAD_STATE_QUERY,                   // after sending query message
BOOTLOAD_STATE_SEND_QUERY_RESP,         // wait to send query response
BOOTLOAD_STATE_INITIATE,                // after send initiate message
BOOTLOAD_STATE_SEND_INITIATE_RESP,      // wait to send initiate response
BOOTLOAD_STATE_RX_IMAGE,               // during receiving OTA data messages
BOOTLOAD_STATE_SEND_DATA,               // during sending OTA data messages
BOOTLOAD_STATE_WAIT_FOR_DATA_ACK,       // wait for ack on data message
BOOTLOAD_STATE_REPORT_ERROR,            // wait to send error message
BOOTLOAD_STATE_SEND_COMPLETE,           // after sending complete message
BOOTLOAD_STATE_WAIT_FOR_COMPLETE_ACK,   // wait for ack on "complete" msg
BOOTLOAD_STATE_VALIDATE,                // after sending validate message
BOOTLOAD_STATE_SEND_VALIDATE_RESP,      // wait to send validate response
BOOTLOAD_STATE_UPDATE,                  // after sending update message
BOOTLOAD_STATE_SEND_UPDATE_RESP         // wait to send update response

};
typedef int8u bootloadState;

// *****************************************************************
// Public functions that are called by the application.

/**@description A function is called in the application's heartbeat or tick
*  function. The function contains basic bootloading state machine and also
*  manages bootload timer.
*
*/
void appBootloadUtilTick(void);
```

```
/**@description Bootload library initialization. Application needs to
* define the ports to be used for printing information and for serial
* bootload.
*  Note that bootload port has to be RS232 port since it needs to receive
*  image via xmodem protocol.
*
* @param appPort: Port used for printing information
* @param bootloadPort: Port used for serial bootloading
* @param appProfileId: Application profile id used
*/
void appBootloadUtilInit(int8u appPort, int8u bootloadPort, int16u
appProfileId);

/**@description A function is called to start bootload process on a target
*  node that is currently running stack/application. The source node sends
*  bootload initiate message to tell the target that it wants to start
*  bootloading. The source node then enters a state waiting for the target
*  node to send an initiate_response, which tells the source node whether
*  the target node is able to participate in the bootload process, along
*  with the error code indicating the reason if it is not able.
*
*/
void appBootloadUtilStartXModem(void);

/**@description A function is called to send query message to gather
*  information about the node(s) with specified address within
EMBER_MAX_HOPS
*  radius. Query message is sent as broadcast. The query is generally used
*  to gather information regarding nodes in the network, especially the
*  eui64 and short id of the node.
*
* @param address: the broadcast address that we want to send to. The valid
* addresses are EMBER_BROADCAST_ADDRESS, EMBER_SLEEPY_BROADCAST_ADDRESS,
* and EMBER_RX_ON_WHEN_IDLE_BROADCAST_ADDRESS.
*
* @param isJIT: specify whether the query message is being sent via JIT
* method or not. The flag is set to TRUE if parent nodes is sending the
* query to the child node when the child polls.
*/
void appBootloadUtilSendQuery(int16u address, boolean isJIT);

/**@description A function is called to start bootload process on a target
*  node that is currently running stack/application. The source node sends
*  bootload initiate message to tell the target that it wants to start
*  bootloading. The source node then enters a state waiting for the target
*  node to send an initiate_response, which tells the source node whether
*  the target node is able to participate in the bootload process, along
*  with the error code indicating the reason if it is not able.
*
* @param target: Node short id to be bootloaded
*
*/
void appBootloadUtilStartBootload(EmberNodeId target);

/**@description A function is called to validate image on EEPROM on a
*  target node. The target node can be a short id of a single node or one of
*  the broadcast addresses. After sending the message, the source node
```

```
* enters a state waiting for the target node to send a validate_response
* with status and image information (timestamp and imageinfo) which tells
* the source node whether the target node has valid image on its EEPROM.
*
*  If the target address is its own, then halAppBootloaderGetImageData()
*  is called directly and no message is sent out.
*
* @param target: Node short id to be bootloaded or broadcast address if
*  wanting to update multiple nodes at the same time.
*
*/
void appBootloadUtilValidate(EmberNodeId target);

/**@description A function is called to install new image (on EEPROM) on a
*  target node. The target node can be a short id of a single node or one of
*  the broadcast addresses. After sending the message, the source node
* enters a state waiting for the target node to send an update_response with
* status which tells the source node whether the target node is able to
* update the image or not.
*
*  If the target address is its own, then halAppBootloaderInstallNewImage()
*  is called directly and no message is sent out.
*
* @param target: Node short id to be bootloaded or broadcast address if
*  wanting to update multiple nodes at the same time.
*
*/
void appBootloadUtilUpdate(EmberNodeId target);

/**@description A function is called in the emberIncomingMessageHander() in
*  order to pass the application bootload message to the library for
*  further processing.
*
*/
void appBootloadUtilIncomingMessageHandler(EmberMessageBuffer message);

/**@description A function is called in the emberMessageSentHandler() in
*  order to pass the sent application bootload message and status to the
*  library for further processing.
*
*/
void appBootloadUtilMessageSent(int16u destination,
     EmberApsFrame *apsFrame,
     EmberMessageBuffer message,
     EmberStatus status);

// ****************************************************************
// Callback functions used by the bootload library.

/**@description A function is called by the library to get the application
*  version and id. MSB returned is the id and LSB is the version.
*
*/
int16u appBootloadUtilGetAppVersionHandler(void);

/**@description A function is called by the library to allow the application
*  to perform any tasks necessary while the node is validating its external
```

```
*  EEPROM image. The validating process can take around ten seconds. Note
*  that while the node is validating EEPROM image, it continues to receive
*  messages over the air. It is strongly recommended that the application
*  calls emberTick() within this function in order to service these
*  messages.
*  Not doing so can lead to the node running out of buffer when it finishes
*  validating the image.
*
*/
void appBootloadUtilValidatingEEPROMImageHandler(void);
```

Application requirements

In order to enable over-the-air application bootloader launch, network node applications must:

- Include app-bootload-utils.h in the application's .h file.
- Include the app-bootload-utils.c file in the project.
- Define app bootload endpoint in emberEndpoints

  Example, `EmberEndpoint emberEndpoints[] = {`

  `{ APP_BOOTLOAD_ENDPOINT, &endpointDescription }`

  `};`
- Define `USE_APP_BOOTLOADER_LIB` in the application's configuration file. Note that app-bootloader-demo sample application also supports bootloading of sleepy (battery power) devices, hence, it has #define `EMBER_APPLICATION_HAS_POLL_HANDLER` to handle querying information from these devices.
- Implement these handlers:

  `int16u appBootloadUtilGetAppVersionHandler(void)`

  `void appBootloadUtilValidatingEEPROMImageHandler(void)`
- Call `appBotloadUtilInit()` after `emberNetworkInit()` which is after `emberInit()` when the application starts.
- Call `bootloadUtilTick()` in a heartbeat function.
- Call `appBootloadUtilIncomingMessageHandler()` in `emberIncomingMessageHandler()`. User needs to verify following parameters before passing the incoming radio messages to the library handler:
  - The application profile ID (assigned to STMicroelectronics and can only be used during development)
  - The cluster id, 0xE002. This defines a set of commands/messages used by this particular application.
  - The endpoint, 240. This is used for application bootloading and is used to filtere incoming radio messages.
- Call `appBootloadUtilMessageSent()` in `emberMessageSent()`. Also the above parameters need to be checked.

The Application bootloader library can be modified to remove functions that are not required to save code space. For example, devices within a network may never perform a serial bootload or initiate bootloading operations. To build an application bootloader library that is a target only node, modify the definitions in app-bootloader-demo-configuration.h as follows:

`USE_APP_BOOTLOADER_LIB` Always defined to build the bootloader demo library

ABL_LIB_TARGET	Define to build a target only node. Node will not be able to perform xModem serial upload, send a query message, initiate a remote bootload, send a validate or update request. Node will be able to receive and store an image OTA and reset to the bootloader to flash the image.

Application Bootloading Process

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. Responding nodes return their application version. The source node evaluates this information and identifies potential target nodes accordingly. The source node queries each potential target node by calling `appBootloadUtilSendQuery()`. This function initiate either a broadcast or unicast message, depending on the argument supplied.
2. On each queried node, the message is passed to `appBootloadUtilIncomingMessageHandler()` and the following information is returned to the source node: eui64 address, device type, application version, mfgId, hwtag, app bootloader version and recovery image version. On the source node, on each received query response, it'll call `emberGetSender()` to obtain the queried node's short id and display it along with its other information.
3. Depending on the query results, the source node can send a bootloader initiate message to a target node by calling `appBootloadUtilStartBootload()` with the node short id as argument.
4. On receiving the initiate message, the target node determines whether the application should allow the request. The node will send initiate response message back with status `APPBL_READY` if it is ready to be bootloaded, otherwise, it will use other status. After sending the response, it will also change its state to wait for incoming bootload message.
5. The source node start sending bootload data only if initiate response contains `APPBL_READY` status. Before sending data, the source node will first validate its image on EEPROM to ensure that it has a valid image. If it doesn't have a valid image or if it later decides not to go through with the bootloading, it will not send bootload data. The target node will then naturally timeout and come out from the waiting state.
6. However, if the bootload process is continued, the source node will send bootload data to the target node using APS retry and enable route discovery option. The target node receives the message and writes the data to its eeprom. The stack will handle the acking and retrying of the message for both source and target nodes. Source node waits until each data message is properly acked before sending the next one. If there is any error, for example, the data is corrupted, target node fails to write to its eeprom or source node does not receive ack to its data message, the node who discovers the error will send an error report message to the other node and the bootload process will stop with a failed status.
7. However, if the bootloading process has been successful so far, after sending the last data message, source node will send a bootload complete message to the target node to signal the end of the process.
8. The source node now can send a validate message to the target node by calling `appBootloadUtilValidate()` to tell the target node to validate its eeprom content. The target node will reply back with validate response which will contain the image information if it successfully validates its image.

9. The source node now can choose when it wants the target node to update its flash image with the newly bootloaded eeprom image. The souce node can update the target node's image by calling `appBootloadUtilUpdate()` with the target node's short id as an argument. After receiving a update image, the target node replies back with update response status. If it is ready to update its image, it will send the response with status `APPBL_UPDATE` then it will call `halAppBootloaderInstallNewImage()` which will copy the content from eeprom to flash.

Library features and limitations

The highlighted features of the application bootloader utility library are:

- The library supports multi-hop downloads. Since all application bootload messages are ZigBee aps unicast messages, they can be routed over multiple hop of nodes.
- The library code that is not used will be deadstripped out. Therefore, in a network, some nodes may act as bootload servers and some may act as bootload clients. For bootload servers, they must to be able to receive bootload messages via xmodem on a UART interface, query, start bootload process, validate, and update the client nodes. On the other hand, the client nodes do not need to have the previously mentioned capabilities, hence, those functions will be stripped for the client node.

The following are constraints of the existing library:

- The library code takes over the serial port on the source node during serial uploads. The bootload baud rate used is 115200 kbps. 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.

Manufacturing tokens

The bootloader does not require that you set the manufacturing tokens shown below; however, customers are welcome to do so. Unlike standalone bootloading, no security handshake is required to start the bootloading process, because that is all handled in the application/stack level. The customers have the option of turning on ZigBee security which will encrypt all over-the-air messages.

- 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 `appBootloadUtilSendQuery` to request to gather information regarding the nodes in the network. The application bootload library code reads the token from simeeprom and includes the information in the query response.

- `TOKEN_MFG_MANUF_ID`

  – A 16-bit (2-byte) string that identifies the manufacturer. Applications can query nodes to obtain their manufacturer ID, and decide whether to bootload a node accordingly. When a node calls `appBootloadUtilSendQuery` to request to gather information regarding the nodes in the network, the application bootload library code reads the token from simeeprom and include the information in the query response.

  – 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 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).

8.4.11 Application bootloading sample application

EmberZNet provides a sample application to demonstrate the use of the application bootloader on development kit devices. This application bootloader demo shows how to integrate the application bootloader utility library into an application. It shows how to use a simple command interface to perform a bootloading process. You can use the application bootloader demo as a reference code example to be customized for updating device images over the air.

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

Serial baud rates and ports used

The original sample application uses a serial port to communicate with users. All commands are transmitted and received via serial port 1. The same port is configured to be the bootload port at 115200 bps when uploading image onto external EEPROM. User is requested to adapt the sample application for matching its platforms requirements.

```
// Commands used for bootloading and forming network
"form"            "Form a network
"join"            "Join the network as router"
"join_sleepy"     "Join the network as sleepy node"
"join_mobile"     "Join the network as mobile node"
"join_end"        "Join the network as end device (non-sleepy)"
"leave"           "Leave the network"
"query"           "Obtain information on all node"
"xmodem"          "Obtain data via xmodem and store in eeprom"
"bootload"        "Transfer data to target node"
"validate"        "Validate (eeprom) image on target node"
"update"          "Update image on target node"
"help"            "List available commands"


// Commands used during debugging.
"status"          "get network status of the node"
"reboot"          "reset the node"
"network_init"    "initialize network stack after a reset"
"set_channel"     "change network channel"
"set_pan_id"      "change network pan id"
"set_epan_id"     "change network extended pan id"
"set_power"       "change transmit power"
"set_tune"        "set node's tune"
"buffer"          "obtain number of free packet buffers"
```

Usage notes:

To start transfer image to the node's eeprom using xmodem protocol over serial port 1, issue command "`xmodem`", the node will signals the event by printing a stream of C characters to

serial port 1. Then select the .ebl file to download using a terminal emulator program (such as HyperTerminal) connected to the node's serial port.

8.4.12 Application bootloader message formats

The following sections describe the purpose of each packet and when it gets used. Note that all application messages are ZigBee unicast messages that contain different aps payload values. The section below describes the difference of the (aps) payload contents for each message. Moreover, application bootload messages are built and parsed using zcl (ZigBee Cluster Library) library utility (app/util/zcl). Hence, each payload (bootload message) contains 5 bytes zcl header.

Table 43. Query message

Bytes	Field	Description/Notes
5	ZCL Header	
2	APP_BL_EUI64_ATTRIBUTE_ID	attribute to be read (0x0000)
2	APP_BL_APPBL_VERSION_ATTRIBUTE_ID	attribute to be read (ox0006)
2	APP_BL_REC_IMAGE_VERSION_ATTRIBUTE_ID	attribute to be read (ox0007)
2	Senders Short Address	Original query sender's short address

Table 44. Query response message

Bytes	Field	Description/Notes
5	ZCL header	
2	APP_BL_EUI64_ATTRIBUTE_ID	attribute to be read (0x0000)
1	read status	success or failure of read
1	attribute type	IEEE address
8	EUI64 address	IEEE address of queried node
2	APP_BL_APPBL_NODETYPE_ATTRIBUTE_ID	attribute to be read (0x0001)
1	read status	succcess or failure of read
1	attribute type	int8u
1	node type	router, end device, mobile or sleepy node
2	APP_BL_APPBL_VERSION_ATTRIBUTE_ID	attribute to be read (0x0002)
1	read status	succcess or failure of read
1	attribute type	int16u
2	application version	application defined
2	APP_BL_APPBL_MFGID_ATTRIBUTE_ID	attribute to be read (0x0004)
1	read status	succcess or failure of read
1	attribute type	int16u

Table 44. Query response message (continued)

Bytes	Field	Description/Notes
2	manufacturer's ID	assigned by ZigBee
2	APP_BL_APPBL_HWTAG_ATTRIBUTE_ID	attribute to be read (0x0005)
1	read status	success or failure of read
1	attribute type	octet string
16	manufacturer's hardware ID	assigned by manufacturer
2	APP_BL_APPBL_VERSION_ATTRIBUTE_ID	attribute to be read (0x0006)
1	read status	succcess or failure of read
1	attribute type	int16u
2	application bootloader version	assigned
2	APP_BL_APPBL_REC_IMAGE_VERSION_ATTRIBUTE_ID	attribute to be read (0x0007)
1	read status	succcess or failure of read
1	attribute type	int16u
2	Recovery image version	assigned

Table 45. Initiate message

Bytes	Field	Description/notes
5	ZCL Header	
2	APP_BL_STATUS_ATTRIBUTE_ID	current bootload status of target node, attribute to be read (0x0008)

Table 46. Initiate response message

Bytes	Field	Description/notes
5	ZCL Header	
1	BL_STATUS	APPBL_READY (0x00) if ready to be bootloaded

Table 47. Data message

Bytes	Field	Description/notes
5	ZCL header	
2	APP_BL_BLOCK_NUMBER_ATTRIBUTE_ID	over the air data block number
2	Block number	
2	APP_BL_DATA_ATTRIBUTE_ID	

Table 47. Data message

Bytes	Field	Description/notes
1	Date length	generally 64 bytes, but can be changed
64	Data	bootload data

Table 48. Report message

Bytes	Field	Description/notes
5	ZCL Header	
2	APP_BL_STATUS_ATRIBUTE_ID	
1	Bootload status	APPBL_COMPLETE if bootload is successful

Table 49. Validate message

Bytes	Field	Description/notes
5	ZCL Header	
2	APP_BL_STATUS_ATTRIBUTE_ID	attribute to be read (0x0008)
2	APP_BL_TIMESTAMP_ATTRIBUTE_ID	attribute to be read (0x0003)
2	APP_BL_IMAGE_INFO_ATTRIBUTE_ID	attribute to be read (0x000B)

Table 50. Validate response message

Bytes	Field	Description/notes
5	ZCL header	
2	APP_BL_STATUS_ATTRIBUTE_ID	attribute to be read (0x0008)
1	read status	success or failure of read
1	attribute type	int8u
1	bootload status	APPBL_IMAGE_VALID if image is valid, otherwise, APPBL_IMAGE_INVALID
2	APP_BL_TIMESTAMP_ATTRIBUTE_ID	attribute to be read (0x0003)
1	read status	success or failure of read
1	attribute type	int32u
4	EEPROM image timestamp	
2	APP_BL_IMAGE_INFO_ATTRIBUTE_ID	attribute to be read (0x000B)
1	read status	success or failure of read
1	attribute type	octet string
32	EEPROM image information	

Table 51. Update message

Bytes	Field	Description/notes
5	ZCL header	
2	APP_BL_STATUS_ATTRIBUTE_ID	attribute to be read (0x0008)

Table 52. Update response message

Bytes	Field	Description/notes
5	ZCL Header	
1	Bootload status	APPBL_UPDATE if the node is ready to update its image

9 Token system

9.1 Introduction

A token is an abstract data constant that has special persistent meaning for an application. 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 will be clear from the context which meaning should be used. In this document we will always be specific: token will always refer to the key + data pair.

Tokens are typically stored in NVRAM (EEPROM, Flash, or other non-volatile memory) so that the token data will persist across reboots and during power loss.

9.1.1 Purpose

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. In the case of EmberZNet, the tokens are used for the STM32W108 implementation, each of which has a different underlying NVRAM mechanism.

The STM32W108 chip uses a special memory-rotation algorithm is used to prevent premature overuse of the underlying Flash. However, the application designer does not need to know any of the underlying details. By using the token key to store and retrieve token data, all of the underlying details may be put aside.

9.2 Usage

EmberZNet provides a simple set of APIs for accessing token data. The full documentation may be found in the EmberZNet HAL API documentation.

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 other 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 will be 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.

Next we will look at some examples of using these types of tokens, and then take a look at how to define custom tokens.

9.3 Standard (non-indexed) tokens

Some applications may need to store configuration data at installation time. Usually, this is a good time to consider using a standard token. If I've defined this token to use the token key DEVICE_INSTALL_DATA, and if I have a data structure that looks like this:

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

Then I 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);
```

Don't worry, we'll explain how to configure your own custom tokens on the next page.

9.3.1 Indexed tokens

Perhaps in addition to storing configuration data, I also wish to store a set of similar values. For example, an array of preferred temperature settings throughout the day. In this case, I may simply use the default data type int16s to store my desired temperatures, and I will 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 my application code, I 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);
  }
```

```
}
```

9.4 Counter tokens

If the other two types of tokens made sense, counter tokens will be very easy. Perhaps I have a token that I wish to use to count the number of heating cycles a thermostat has initiated -- that would be a perfect use for a counter token. I name it `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;
}
```

If this all makes sense, and you're waiting to find out how to define your own tokens just like this, read on...

9.5 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`. More information about preprocessor variables and how they can be configured is available.

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 does 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.

9.5.1 Mechanics

Adding a custom token to this file is easy. It 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.

We will define the token examples from the previous page to give a good idea of how to use each of the three types of tokens.

Define the token name

We need three token names. When defining the name, we 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 is defining the token key and linking 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 we refer to them 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 of your custom tokens should have a token key less than 0xA000.

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 will guarantee a proper link between application and data.

Define the token type

Each token in this case 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.

We define it the same way as in the previous example. Note that it must only be defined 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 room_number; /** The room where this device is installed */
  int8u install_date[11] /** YYYY-mm-dd + NULL */
} InstallationData_t;
#endif //DEFINETYPES
```

Define the token storage

This part is the key to tying it all together. This actually informs the token management software about the tokens you are defining. Each token 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
```

This is a little complicated, so let's dissect it piece by piece:

```
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; while it may appear complicated, many examples are floating around on the web. In this case, the first value (`room_number`) is initialized to 0, and the next value (`installation_date`) is set to all 0's because the {0,...} syntax fills the remainder of the array with 0.

The other two definitions should now make sense. The only difference is that `DEFINE_INDEXED_TOKEN` requires, as you might expect, a length of the array -- in this case, `HOURS_IN_DAY`, or 24. The final argument to it is the default value of every element in the array. Again, in this case it is initialized to all 0.

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

The stack defines default tokens and we discuss them here, and touch on some advanced topics that are not covered in detail in this tutorial, such as fixed tokens and setting default tokens. Resources are suggested for further information.

9.6 Default tokens

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

- Stack Tokens, which are runtime configuration options set by the stack; these should not be changed by the application
- Manufacturing Tokens, which are set at manufacturing time and cannot be changed by the application

The information in this depends on the version of EmberZNet you are using. To find the information for your version, please view the file <install-dir>/stack/config/token-stack.h. Search for CREATOR to see the defined names. Note that this file may be quite confusing for first-time viewing -- if it is overwhelming, just focus on that section.

9.6.1 Stack tokens

These are used by the stack internally. They should not be modified by the application.

```
CREATOR_STACK_NVDATA_VERSION (0xFF01)
CREATOR_STACK_BOOT_COUNTER (0xE263)
CREATOR_STACK_NONCE_COUNTER (0xE563)
CREATOR_STACK_ANALYSIS_REBOOT (0xE162)
CREATOR_STACK_KEYS (0xEB79)
```

```
CREATOR_STACK_NODE_DATA (0xEE64)
CREATOR_STACK_CLASSIC_DATA (0xE364)
CREATOR_STACK_ALTERNATE_KEY (0xE475)
CREATOR_STACK_APS_FRAME_COUNTER (0xE123)
CREATOR_STACK_TRUST_CENTER (0xE124)
CREATOR_STACK_NETWORK_MANAGEMENT (0xE125)
CREATOR_MFG_EZSP_STORAGE (0xCD53)
CREATOR_MFG_ASH_CONFIG (0xC143) // msb+'A'+'C' (ASH Config)
CREATOR_STACK_CAL_DATA (0xD243) // msb+'R'+'C' (Radio Calibration)
```

9.6.2 Manufacturing tokens

These are used by the stack internally and cannot be modified by the application. Some of them can be set at manufacturing time (see below); others come pre-programmed by STMicroelectronics

Note: These files are actually chip dependent, and are referenced from token-stack.h. These entries are from hal/micro/cortexm3/token-manufacturing.h file

```
CREATOR_MFG_CHIP_DATA (0xC344)
CREATOR_MFG_NVDATA_VERSION (0xFF09)
CREATOR_MFG_EMBER_EUI_64 (0xE545)
CREATOR_MFG_TRIM_DATA (0xD444)
CREATOR_MFG_CUSTOM_VERSION (0xC356)
CREATOR_MFG_CUSTOM_EUI_64 (0xE345)
CREATOR_MFG_STRING (0xED73)
CREATOR_MFG_BOARD_NAME (0xC24E) // msb+'B'+'N' (Board Name)
CREATOR_MFG_OSC24M_BIAS_TRIM (0xB254) // msb+'2'+'T' (2[4mHz] Trim)
CREATOR_MFG_EUI_64 (0xB634)
CREATOR_MFG_MANUF_ID (0xC944) // msb+'I'+'D' (Id)
CREATOR_MFG_PHY_CONFIG (0xD043) // msb+'P'+'C' (Phy Config)
CREATOR_MFG_BOOTLOAD_AES_KEY (0xC24B) // msb+'B'+'K' (Bootloader
Key)
```

Manufacturing 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 STMicroelectronics. Other tokens, such as the internal EUI-64, cannot be overwritten.

For more information about manufacturing and token programming, refer to Bringing Up Custom Devices for the STM32W108 SoC Platform (contained in the documentation/ApplicationNotes directory of the EmberZNet stack installation).

Fixed tokens

The token utility allows tokens to be placed at fixed locations, and these tokens use the `DEFINE_FIXED...TOKEN()` macros. In general these are useful only for manufacturing tokens and their use in applications is discouraged; in any case, their use should be carefully considered and used only if necessary.

More information about fixed tokens can be found in the token.h header file, <install-dir>/hal/micro/token.h, as well as the HAL API guide (HTML-formatted and possibly more readable).

9.7 Bindings

Tokens can be involved in transport layer messaging and, like other devices, must be included in a binding table. This table can reside in RAM, EEPROM, or some combination of both and has a configurable number of entries. 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, refer to the online EmberZNet API documentation for `EMBER_BINDING_TABLE_SIZE` and to the online HAL documentation for `TOKEN_BINDING_TABLE`.

Table 53 lists token related static memory constants that manage RAM. These are set in stack/include/ember-configuration-defaults.h, and can be reset through preprocessor definitions at compile time.

Table 53. Static memory defines for tokens

Constants	Description

`EMBER_BINDING_TABLE_SIZE`	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.
	A binding is needed for any message destination (multicast group or unicast point), but it is a good idea to dynamically remove any bindings that are unused for a period of time, and thus conserve runtime RAM. You can also save RAM by storing nonvolatile or often used bindings in EEPROM by using the `BINDING_TABLE` token. Temporary binding entries each use 2 bytes of RAM. The rest of the binding table (size minus temporary) is assumed to reside in EEPROM. The default binding table size is 8 entries, all RAM-based. The following errors indicate binding table size problems:
	`EMBER_NULL_BINDING`
	`EMBER_BINDING_INDEX_OUT_OF_RANGE`
	`EMBER_INCOMPATIBLE_STATIC_MEMORY_DEFINITIONS`

9.8 For more information

For more detailed information about the underlying token mechanism and the simulated EEPROM that provides the STM32W108 NVRAM implementation, please refer to the HAL API guide, which has detailed information about the use of these items.

10 Testing and debug strategies for ZigBee application development

10.1 Introduction

There are a number of shipping ZigBee products now in the marketplace and evaluation of their development and testing strategies can assist other OEM's. Development and testing of embedded applications has always relied on specific testing and debug strategies such as the use of emulators or JTAG to determine what is detect and isolate problems. This has worked well when a problem exists on a single device or between two devices communicating. Commercial development and deployment of larger distributed embedded network applications has required another level of testing strategies. Without planning of the testing and debug process, development projects can stall in a cycle of bug fixing, field testing, bug fixing and field testing until it is hoped all problems have been resolved. A series of specific testing strategies throughout the development process are recommended. The development strategy and testing processes required are similar even if the products being developed are aimed at different marketplaces.

This chapter will review the typical testing methodology that has been successful and the hardware and software tools required for qualification of products. This paper will provide a testing outline and methodology that can be used for software qualification. Consideration of this testing methodology is critical from the beginning of the development efforts to ensure suitable means for qualification are built into the software including appropriate means for simulating testing and recording test results. Real world examples and test setups will be used to illustrate the testing strategies proposed in this paper. The key areas of testing to be covered will be:

- Hardware and Application considerations for testing and debug
- Initial development and lab testing
- Beta criteria and field trials
- Release testing process and criteria for release

10.2 Hardware and application choices for testing and debug

10.2.1 Initial software application development using development kit hardware

Initial development of customer applications can typically start using development kit hardware that is available from the chip and software supplier. These kits universally have some level of serial or Ethernet debug capabilities to allow viewing what is occurring at the application as well as at the ZigBee software stack. It is important for a developer to start with these tools to evaluate what information is available and how it can be used in the development, debug and testing.

Some typical choices to be made early include:

- Will the software provide debug information out a debug port or serial port for later use in debug and testing? Such information can be critical to evaluating problems later and significantly shorten the debug process. However, use of a serial port may slow the application and impact normal operation. A serial port may also not be available during

normal system operation. Even if not normally available, a serial or debug port can be populated on prototype hardware for early stages of testing.

- How will software be monitored and upgraded during internal testing? It is required to regularly update software during development and initial testing. For these systems it is recommended that a means of rapidly upgrading software on all devices be included even if it is not included on final field hardware. For example, on STMicroelectronics development kits, the dedicated debug and programming port can be accessed over a USB Virtual com. During initial development and testing, use of this wired backchannel is recommended for both software upgrading and system monitoring.
- How will software be monitored and upgraded during field testing? Initial field testing will also result in periodic updating of software. However, mechanisms used during internal testing may be impractical during actual field testing. An over the air (OTA) bootloader is recommended for field testing for software upgrading. A debug channel is impractical for all devices during field testing. Instead, it is recommended that selected devices such as gateways be monitored with a debug channel and sniffers be used to monitor the network.
- What mechanisms will be provided for system testing and qualification? It is important to be able to trigger system level events and normal operations during qualification testing. This can be done manually by operating the system or it can be done using backchannel or serial port mechanisms but it must be considered early in the design.

Initial application development and testing efforts on development kit hardware should consider each of these questions. Depending on the particular hardware being design, it may or not be practical to include all of the recommendation above but they should be considered.

10.2.2 Transition to custom hardware

Most customer move to the specific designed hardware for their application as soon as practical. However, the specific limitations and constraints of the hardware design may make it impractical to include monitoring and debug ports. At a minimum, tests points should be included to allow access when required.

If monitoring and debug ports are not practical on the custom hardware, it is recommended that development continue on both customer hardware and the development kit hardware so full debugging capabilities are available on at least some of the hardware. For STMicrolectronics chips the debug port is also the programming port and this should be available on initial hardware designs since devices will typically be programmed numerous times during development and debug.

Many networks have a centralized base station, gateway or controller device. This device plays a critical role in the network and therefore plays a critical role in monitoring and debugging the network. If other devices cannot have a debug and monitoring port it is important this device include one.

It should be noted that the questions about debug access and programming must be considered on the customer hardware during the design phase to avoid a hardware respin later to add these capabilities. Software engineers should review schematics to ensure the capabilities provides in hardware match those required for the software development and debugging.

10.3 Initial development and lab testing

10.3.1 Initial development environment and system testing

Initial development and testing is typically done on a desktop or benchtop environment to ensure a basic application is operational prior to expanding to a larger network.

Debug library

The first thing that should be done is to link the EmberZNet Debug Library into your compiled application. ember-debug.h includes all of the debug functions useful during development. A full description of the library can be found in the online EmberZNet Stack API Reference. Once the application has met all design goals, the debug library can be excluded from the final build.

Single device testing and debug

One of the initial steps in starting a new application is to isolate the network down to several devices and initially focus on the message flow and application logic. This may be as simple as light and a switch or a controller and a single device. This testing validates the simple messaging protocols.

For devices with very specific and time critical operations, debugging on the single device to ensure proper operation is necessary. This is common on devices doing lighting control or dimming where precise timing is needed to maintain lighting levels. In these cases it is important to develop and debug the application using specific debug tools connected to the individual device. Once this testing is completed, it is important to note that timing may be effected by operation of the network stack and therefore this testing needs to be also done during network level testing to verify proper operation of the individual device.

System test scenarios

Once single device testing and debugging is completed, more directed system level testing is required. This directed testing should reflect the expected network operating environment and conditions. This testing is intended to specifically ensure expected operations and network conditions is replicated in the test environment where problems can be uncovered and resolved.

For any system level testing, it is important to define the expected operation under specific network and application scenarios. These include the following:

1. System Start Up - The expected normal system start up and commissioning should be validated and tested at each step of the expanding testing below.
2. Typical System Operation - The expected normal system operation and data flow through the network should be tested to ensure smooth and consistent operation.
3. Security - Many applications run extensive lab and field testing prior to turning on security. The use of security can make troubleshooting and debug more difficult depending on the sniffer tools used. It is important to have a debug system that decrypts the packets prior to display. Use of security impacts the payload size available for the application, and increases system level latency due to more node processing time. The impact of security on system level performance needs to be identified early in the system level testing.
4. Stressed System Operation - If the system has particular devices that generate messages based on user interaction of system behavior, these items should be tested

well beyond normal operation to understand the behavior when the system is more stressed. The acceptable behavior under these conditions must be defined and then tested. For example, in some systems it may be acceptable to discard a message under high traffic conditions and send another message later. In other systems the message should be retried by the application.

5. Power Failure and Restart - It is expected that systems may lose power either partially or totally due to a power outage or building maintenance. The system has to react and recover from power failure and resume operations. Battery operated sleeping devices must conserve power during this loss of network connectivity and then rejoin the network once it restarts. The expected time of the restart will vary based on the application design and use of sleeping devices. The behavior of the application on this restart should be defined and tested.
6. Performance Testing - Many applications have specific requirements for end-to-end latency, delivery reliability or other system criteria. These specific metrics should be defined and a means for clearly measuring the metrics during testing should be determined.
7. Typical Application Failure Cases - For any system, the specific failure mechanisms may be different. However, typical failure cases should be identified and added to the test plan for each stage of development. Typical scenarios include:
   - low battery on end device
   - loss of end device
   - loss of parent (child fail over to new parent)
   - loss of routers in network
   - loss of gateway or base station

System level test networks

In addition to the test scenarios above, dedicated test setups are necessary to establish the proper conditions expected in the field. In our experience, ad hoc testing does not uncover all the expected use cases and therefore directed testing to force these network conditions is required.

In each of these test setups, it is not necessary to run directed or specific tests. However, it is important to ensure the full range of typical expected operating conditions are exercised.

- Multi-hop test network

  Actual field conditions are not always known, and actual radio range under the expected installed environment are not known. It is important to ensure a multihop test environment is established. Sometimes testing is done on an ad-hoc basis with desktop or office networks but more than one hop is not checked and tested. One method we have used to ensure testing of a multihop network is building of a wired test network with splitters and attenuators in the RF path. This provides a repeatable test environment for regression testing. This test environment includes RF cabling, connectors, attenuators and splitters. Each splitter can represent 3 nodes at a particular network depth and the signal is then attenuated to the next splitter that forms the next hop in the network. A network can then be built of as many hops as expected. This test should be built to replicate as many hops as expected in the field installations, or at least 5 hops.
- Dense network

  Dense networks often can present a challenge as the network has to deal with increase in table sizes and message flow. The application also experiences more delays in

message flow since all radios within range must share time on the air. It is recommended that at least 18 nodes be included in this testing.

- Mobile device testing

  Many system designs have one or more mobile devices. These may be remote controls, handheld devices or operator indication devices. Testing of the mobility of such devices and their ability to reconnect to the network is an important test prior to field testing. Problems with mobile devices can be difficult to track in the field because the device is moving around the network and therefore harder to troubleshoot. Simple mobile node testing could be added to the existing wired test installations with a switch cable attenuator under control of a application. This allows moving of the device from one side of the network to another and back again.

- Large network testing

  After the above dedicated tests, software can then be testing in a large scale test network.

- Coexistence and interference testing

  Testing also considered the expected environment and other wireless systems that may be operating. Almost any type of typical installation is likely to have to share the 2.4 GHz band with Wifi or 802.11 type traffic, Bluetooth, cordless phones, baby monitors and the many other devices that share the 2.4 GHz band. Depending on the expected field conditions, system level testing should consider including any of these devices during the testing.

10.4 Moving to beta and field trials

10.4.1 Hardware and test system for larger system testing

Beyond the more dedicated test systems discussed above, it is necessary to move to a wider system level test environment within development and then within initial field testing.

While a stack provider Ican have dedicated setups for various conditions, as well as a larger test network with complete debug connectivity, this is not always practical for the typical application developer. Instead, the developer must focus on realistic tests that can be setup and operated within the budget and space constraints of their project.

First and most importantly, it is usually not practical to have full backchannel analysis capability for a wider scale system on custom hardware. This does not mean the developer should give up on the use and analysis of debug data, it has to be acquired in different means. Several techniques we have seen used:

1. Full Sniffer Based Environment - If the custom hardware does not have any access for debug data, dedicated sniffers can be used. To be useful they should be spread throughout the system to capture as much data as possible back to one central log file.
2. Partial Sniffer and Debug Data - If some of the custom hardware has accessible debug ports, a hybrid system of some debug ports and additional sniffers to fill out the network picture can be used. This is especially valuable if there are particular central control devices that can be monitored using debug while more remote areas of the network are monitored using sniffers.
3. Full Debug Environment - If problems are seen with either of the above setups, the stack provider can be requested to operate the customer application in there more dedicated test network with full debug access. While this involves porting the

application to development hardware, it does provide full debug capabilities for more difficult problems.

For any of the test environments used for the full system testing, there are several critical items that need to be included for proper testing and debugging. These include:

1. Mechanisms to activate typical application level operations. Through scripting or other means it is important that the full system setup provide a means to force application level actions so the system can be repeatably tested. A problem that is seen once but cannot be replicated because of an uncontrolled test environment is difficult to resolve.
2. Centralized logging of all sniffer and debug information. Where possible it is important to have a central computer that is collecting data into a single log file. This is critical for time stamping of events, duplicate detection and removal and proper analysis of what occurs leading up to a problem.
3. Means for simple upgrading of code in all devices in the network. The best designed system will have bugs and features that require upgrading of software in the test network. Providing a means to upgrade all devices easily provides for a faster development and testing process.

10.4.2 Reproducing common field conditions or problems

Typical field problems occur due to several conditions that are not replicated in testing prior to initial field deployments. Evaluation under expected conditions while still in a controlled test setup instead saves many hours and trips to the field installations later.

Many of these conditions occur under error conditions in the network that were not tested in initial testing.

The most common problems we have experienced in field settings are as follows:

1. Multi-hop Performance - While it sounds unusual, many customers operate on desktop test networks that are generally 1 hop and therefore do not test multi-hop performance until an actual field installation occurs. Adding hops in the network obviously adds routing complexity that the network is intended to handle. However, it also increases packet latency and the response time an application can expect. If the application cannot handle this higher latency then message failures or excessive retries occur in the field.
2. Density or Sparsity of Network - Desktop testing of units, even those units that are on the final field hardware design, often does not replicate the actual field conditions that are expected. Often devices are installed in walls or ceilings, within metal enclosures, with metal panels or shields over the device, or more widely spaced than any lab testing. In addition, use of particular network features such as broadcasts or multicasts is impacted by the density or sparsity of the network.
3. Required Application Interrupt Timing Under Network Loading - Some applications require tight control over interrupt latency to perform critical functions. However, the networking stack also requires interrupt level processing on receiving of messages. While the network has the ability to buffer messages, this buffering is limited based on the memory of the device. Under actual network conditions, nodes that run out of buffer space temporarily lose the ability to send and receive messages. Alternatively, servicing network level interrupts over application level interrupts results in poor application level performance.
4. Excessive Network Traffic - A common complaint is a network that appeared to be operating fine has a decrease in throughput or increase in latency. Upon investigation there is excessive traffic in the network resulting in congestion, lost messages and

degraded performance. The most common reason is ongoing broadcasts or multicasts within the network. Failed messages can then lead to more broadcasted route discoveries leading to further decreased performance.

10.4.3 Initial field deployments

Initial field deployments are a critical step in the development process. These tests validate decisions and testing done throughout the development process and uncover any issues prior to wider deployment. These deployments are normally restricted to several sites and carefully monitored and controlled.

The following items are critical for initial field deployments:

1. Monitoring and Analysis Plan in Place - The initial deployments of a field system require the ability to monitor and analyze the system performance. This can be done at the system level by keeping track of application level failures, or it can be done at a lower level by including some level of sniffers or debug capability in the field network.
2. Clear Operational Criteria and Success Criteria - Prior to installing a field system, the operating criteria and success or failure criteria should be established so these items can be monitored throughout the deployment.
3. Ability to Escalate Problems - Many networks have third party components or systems or critical subsystems such as the networking stack. A clear process for escalation and resolution of bugs is critical to ensure issues are identified and closed in a timely manner.

10.4.4 Release testing and criteria for release

For release of a wireless product, it is important to follow a progressive testing process that starts with desktop testing, lab testing, initial field deployments and release testing. The earlier a problem is identified the simpler it is identify and resolve. It is estimated bugs take 10 to 50 times longer to resolve in a field deployment than in laboratory or office testing.

The release process is an accumulation of the testing discussed above. However, it is typical to have some final test process and validation for final release. This criteria varies among different companies but often includes the following elements:

1. Normal System Operation - Typically running a normal system in a typical environment for some period of time with no failures.
2. Power Cycling and Restart - This testing is done on a repetitive basis to ensure uncontrolled power loss and restarting does not result in failed devices. Typically we have seen this done on a small network with automated power cycling. No failures are allowed.

Table 54. Testing strategies during various development phases

Monitor mechanism	Single Device Development	Desktop Development	Lab Testing	Initial Field Testing	Release Testing
Full debug access	XX	XX			
Partial debug			XX	XX	XX
Sniffer based					XX
Dedicated test types					
High density			XX		XX

Table 54. Testing strategies during various development phases (continued)

Monitor mechanism	Single Device Development	Desktop Development	Lab Testing	Initial Field Testing	Release Testing
MultiHop		XX	XX		XX
Mobile node			XX	XX	XX
Large Network			XX	XX	
Coexistance/ Interference			XX	XX	
System Test Cases					
System Start Up		XX		XX	XX
Normal operation		XX		XX	XX
Security			XX	XX	
Stressed Operation			XX		XX
Power failure and restart		XX			XX
Performance		XX		XX	
Typical failure mechanisms		XX			XX

11 Revision history

Table 55. Document revision history

Date	Revision	Changes
19-Mar-2010	1	Initial release.

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com