AN724: Designing for Multiple Networks on a Single Zigbee Chip

This application note discusses the design considerations involved in integrating a feature that allows a node with one Zigbee radio to operate on multiple Zigbee networks.

Prior to EmberZNet PRO 6.8.0, the multi-network implementation limited the number of always-on roles that a device could serve on the participating networks. Starting with EmberZNet PRO 6.8.0, the multi-network feature and a new multi-PAN feature allow the device to operate as a coordinator on both Zigbee networks in a host-NCP configuration.

KEY POINTS

- Multi-network vs Multi-PAN
- Design considerations
- Network context and stack APIs
- Application framework support
- Tokens
- Multi-network timing considerations

1. Introduction

Multi-network and multi-PAN features allow a device to operate on two Zigbee networks using the same radio. These Zigbee networks may have different security settings or network parameters, such as short ID, PAN ID, extended PAN ID, or radio power. The only parameter that stays the same on all networks is the node's EUI64. While the multi-network feature allows the two networks to be on different radio channels, the multi-PAN feature requires that this setting matches.

This application note presents the software options to implement a feature wherein a device with a single Zigbee radio may concurrently operate on two Zigbee networks. Depending on the intended use case, you can choose either of the following configurations:

- Multi-network: Device may serve in an always-on role on a maximum of one of the two networks. This feature is implemented by timesharing the radio between the two networks. Consequently, the networks may have non-overlapping network and radio settings.
- Multi-PAN: An extension of the multi-network feature wherein a device may serve in an always-on role (specifically a coordinator) on both networks in a host-NCP configuration. The tradeoff is that the radio channel needs to match on both networks. Because this configuration needs to keep track of the state of both networks, there is a significant increase in flash and RAM usage.

Each of these configurations has its own set of limitations. These restrictions and other considerations are discussed in greater detail in the following sections.

2. Design Considerations

There are several factors to consider when designing a node that can operate concurrently on multiple networks as illustrated in the following figure. This section discusses them in detail as well as the associated tradeoffs.

Note: As of this revision of this application note, the number of concurrent networks is limited to two.

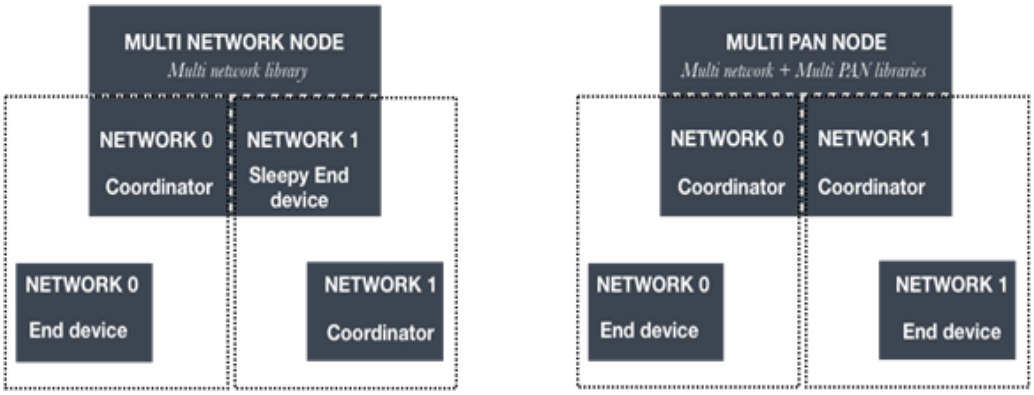

Figure 2.1. Implementation Examples

Based on the roles that the node is expected to serve on the networks, there are three general use cases:

- **Single-network:** This use case is the standard version of the EmberZNet PRO stack where a node may only be part of one network at a time. This configuration links in stub versions of the multi-network and multi-pan libraries. This is often the only configuration that is available on parts that are severely code-space restricted.
- **Multi-network:** This use case is applicable when the Zigbee radio on a device is timeshared between two networks wherein the device serves the role of an always-on device on no more than one network. An always-on device may be a Zigbee PRO coordinator, router, or (non-sleepy) end device.

  To function as an always-on device on one network, the device must limit its role as a sleepy end-device on the other network. The node may also choose to function as a sleepy end-device on both networks. In this case, it can sleep when it is not active on any network. This allows the two networks to be on different radio channels, run at different radio power levels, and have dissimilar network security requirements. This configuration links in the multi-network library and a stub version of the multi-pan library.

  The network scheduling algorithm seamlessly takes care of switching between networks so that the node is on the always-on network unless it explicitly polls or sends some data to the parent on the sleepy end device or sleepy network(s). As soon as the polling or data sending transaction on the sleepy network is completed, the node immediately switches back to the always-on network until a new polling or data sending transaction is initiated. A node that is a coordinator/router on one network and a sleepy end device on the other will not be able to save energy by temporarily shutting down the radio (sleep mode). On the other hand, a node that is a sleepy end device on all networks can enter sleep mode, thus providing energy savings.

  Due to the additional overhead required to maintain information on both networks, there is a nominal increase in code space and RAM usage when compared to the single-network use case. Therefore, this feature is limited to parts with flash size of at least 512kB.

- **Multi-PAN:** This use case is an extension of the multi-network use case where a device with a single Zigbee radio operates two Zigbee networks and takes the role of an always-on node on both networks. This configuration is only supported in Host-NCP mode. Host applications that support two coordinator nodes require an NCP that is similarly configured to support Zigbee stack functionality for the two networks.

  To function as an always-on device on both networks, there is an additional restriction that both networks need to be on the same radio channel so they can receive messages meant for either network at all times. The onus is upon the application to ensure that both networks are on the same radio channel. Starting with EmberZNet PRO 6.8.0, the stack supports a dual-coordinator configuration only. The first network index supports Zigbee 3.0 security and the second network index only supports the no-security Zigbee PRO network configuration profile. The stack does not currently restrict application-initiated channel changes. This is acceptable when only one of the networks is up. However, when both networks are functional, care must be taken to avoid disruptive network or user-initiated channel changes as these are not supported.

  In this configuration, it is also important to note that there is a limitation regarding sleepy broadcast messages. There is a maximum of one broadcast that is queued for all sleepy end devices across both networks.

  This configuration links in both the multi-pan and multi-network libraries. There is more overhead involved in serving as a coordinator compared to a router or an end-device. As a result, this configuration with two coordinators results in the most flash and RAM usage and is limited to parts with a flash size of at least 512 kB.

  Note: Some components are not multi-network-aware and would require additional implementation to enable expected functionality. These include the Device Table component, Update TC Link Key component, and Network Steering component.

Table 2.1. Allowed Role Combinations

Network 0	Network 1	Required Libraries
Zigbee PRO (Coordinator, Router, End device or Sleepy end device)	N/A	Multi-network stub; Multi-PAN stub
ZigBee PRO Coordinator (always-on)	ZigBee PRO Sleepy end device	Multi-network library; Multi-PAN stub
ZigBee PRO Router (always-on)	ZigBee PRO Sleepy end device	Multi-network library; Multi-PAN stub
ZigBee PRO End device (always-on)	ZigBee PRO Sleepy end device	Multi-network library; Multi-PAN stub
ZigBee PRO Sleepy end device	ZigBee PRO Sleepy end device	Multi-network library; Multi-PAN stub
Zigbee PRO Coordinator with Zigbee 3.0 security (always-on)	Zigbee PRO Coordinator with no-security (always-on)	Multi-network library; Multi-PAN library

3. Network Context and Stack APIs

The EmberZNet PRO stack internally manages multiple networks by maintaining multiple network contexts and by switching to the appropriate network context when required. The internal network context at the stack is totally transparent to the application, and the application has no means of interfering with the stack internal network context.

The stack also stores the application's current network context. Every API call invoked in the application code refers to the application current network. For instance, an `emberGetNodeId()` call returns the node's short ID on the network referred by the application current network. The following table lists the APIs provided to set and get the application current network.

Table 3.1. Multi-Network Stack APIs

Ember API	Description
uint8_t emberGetCurrentNetwork()	Returns the index of the current application network context.
EmberStatus emberSetCurrentNetwork(uint8_t index)	Sets the current application current network context.
uint8_t emberGetCallbackNetwork()	Returns the index of the network context of the current callback. If this function is invoked outside of a stack callback, it returns 0xFF.

The first two APIs are straightforward get and set functions for the application network context. The application network context is set by passing `emberSetCurrentNetwork` a network context index. If the network index to this function is invalid, the API returns the error code `EMBER_INDEX_OUT_OF_RANGE`.

Note: The set function is also available as a Command Line Interface (CLI) command `network set <x>`. This allows for a seamless extension of other CLI commands to perform network specific operations. The following is an example on how the set API may be used on a multi-PAN device to form two centralized networks.

```
// Set network context. This serves as application network index for the APIs that follow
// This is similar to CLI command "network set 0"
emberSetNetwork(0);

// Form centralized network on network index 0
emberAfPluginNetworkCreatorStart(true);

// Set network context. This serves as application network index for the APIs that follow
// This is similar to CLI command "network set 1"
emberSetNetwork(1);

// Form centralized network on network index 1
// params.channels must match the single channel from the first network!
emberFormNetwork(params);
```

The `emberGetCallbackNetwork()` API makes the application aware of the network context for the callback. For instance, when the stack calls `emberIncomingMessageHandler()`, the application may invoke `emberGetCallbackNetwork()` inside the callback to retrieve the index of the network on which the packet was received. The following is an example on how to query the network context for a handler.

```
void emberIncomingMessageHandler ( EmberIncomingMessageType type,
      EmberApsFrame *apsFrame,
      EmberMessageBuffer message)
{
   //Get network index for incoming packet
   uint8_t nwkIndex = emberGetCallbackNetwork();

   emberAfPushCallbackNetworkIndex();
   // Perform any network specific operations here
   emberAfPopNetworkIndex();
}
```

4. Zigbee Application Framework Support for Nodes on Multiple Networks

The Zigbee application framework provides additional network management functionality to help you easily build and deploy devices that operate on multiple networks. For more information on implementing multi-network functionality with the Zigbee application framework, see *UG491: Zigbee Application Framework Developer's Guide for SDK 7.0 and Higher*. The Zigbee application framework provides many advantages to the application developer in terms of reduced complexity, mostly relating to how it seamlessly manages the different network contexts when sending and receiving messages, before invoking callbacks, and before triggering event handlers for certain types of events.

The Zigbee application framework allows users to configure options such as the Zigbee device type and security for the individual networks on the Zigbee stack. When operating a device concurrently on two Zigbee networks in a host-NCP configuration, you must choose a similarly capable NCP that matches the host configuration. Multi-network library and multi-pan library components are provided to enable and implement the appropriate functionality on the Zigbee stack.

The EmberZNet PRO SDK comes with the following sample applications that demonstrate one potential configuration. You can use these as a reference to make a different application while being mindful of the limitations. Refer to section 2 Design Considerations for additional guidance.

- Zigbee - Host MpZ3TcCustomTcHost: A host application that implements a node which is able to serve as a coordinator on two networks. The first network uses Zigbee 3.0 security and the second network uses no-security. You can use this host application in combination with either of the following NCP sample applications.
- Zigbee - NCP mp-ncp-uart-hw: A multi-PAN aware NCP application that communicates with the host over UART with hardware flow control.
- Zigbee - NCP mp-ncp-spi: A multi-PAN-aware NCP application that communicates with the host over SPI.

4.1 Handling of Different Network Contexts

- **Endpoint mapping:** For Zigbee PRO networks, the Zigbee application framework maintains a mapping between network contexts and disjoint sets of endpoints. For example, network 0 may be associated with the set of endpoints {1,2,3} while network 1 is associated with the set {4,5,6}. Notice that these two sets of endpoints must be disjoint; that is, the same endpoint cannot be associated with two different network contexts. For each application framework API that is endpoint-related, the framework switches network context according to the endpoint mapping. For instance, when we send a unicast message using the application framework API `emberAfSendUnicast()`, the application framework looks into the source endpoint specified in the passed `apsFrame struct` and switches to the corresponding network context prior to passing the packet to the stack. The network context is then restored to the original one.
- **Callbacks and network-specific handlers:** The Zigbee application framework ensures that the network context is always correctly set when an application framework callback or network context-sensitive event handlers are invoked. For example, when `emberAfPreMessageReceivedCallback()` is called, the current network context gets set to that of the receiving network. Note that framework sets the network context for callbacks with the `emberAf` prefix, but does not set the context for handlers called by the stack. For example, the network context will not be set automatically when the stack calls `emberPollCompleteHandler()`. However, the End Device Support plugin will set the appropriate context before calling `emberAfPluginEndDeviceSupportPollCompletedCallback()`. The following example provides sample code to query the network index in an application framework callback.

```
void emberAfPluginEndDeviceSupportPollCompletedCallback(EmberStatus status)
{
  //Get network index for end device support poll complete handler
  uint8_t nwkIndex = emberGetCallbackNetwork();

}
```

4.2 Multi-Network Application Framework APIs

The application framework provides a set of APIs for dealing with network context. These APIs are used in the application framework code to perform network context switching as described above. Notice that in all the code provided by the application framework, network context switching is performed using the APIs in the following table.

Table 4.1. Multi-Network Application Framework APIs

API	Description
EmberStatus emberAfPushNetworkIndex(uint8_t index)	Sets the current network to that of the given index and adds it to the stack of networks maintained by the framework. Every call to this API must be paired with a subsequent call to emberAfPopNetworkIndex().
EmberStatus emberAfPushCallbackNetworkIndex(void)	Sets the current network to the callback network and adds it to the stack of networks maintained by the framework. Every call to this API must be paired with a subsequent call to emberAfPopNetworkIndex().
EmberStatus emberAfPushEndpointNetworkIndex(uint8_t endpoint)	Sets the current network to that of the given endpoint and adds it to the stack of networks maintained by the framework. Every call to this API must be paired with a subsequent call to emberAfPopNetworkIndex().
EmberStatus emberAfPopNetworkIndex(void)	Removes the topmost network from the stack of networks maintained by the framework and sets the current network to the new topmost network. Every call to this API must be paired with a prior call to emberAfPushNetworkIndex(), emberAfPushCallbackNetworkIndex(), or emberAfPushEndPointNetworkIndex().
uint8_t emberAfPrimaryEndpointForNetworkIndex(uint8_t index)	Returns the primary endpoint of the given network index or 0xFF if no endpoints belong to the network.
uint8_t emberAfPrimaryEndpointForCurrentNetworkIndex(void)	Returns the primary endpoint of the current network index or 0xFF if no endpoints belong to the current network.
uint8_t emberAfNetworkIndexFromEndpoint(uint8_t endpoint)	Returns the network index of a given endpoint or 0xFF if the endpoint does not exist.
uint8_t emberAfNetworkIndexFromEndpointIndex(uint8_t index)	Returns the network index of the endpoint corresponding to the passed index or 0xFF if no endpoint is currently stored at the passed index.

5. Non-Volatile Storage Tokens

Network-related tokens are duplicated for each participating network. Tokens that pertain to a network may be accessed using the standard APIs after setting the application network context. The tokens that have been duplicated are summarized in the following table.

Table 5.1. Tokens that are Duplicated for Each Participating Network

Node data (node ID, node type, PAN ID, extended PAN ID, tx power, channel, profile)
Network key and network key sequence number
Network alternate key and key sequence number
Network security frame counter
Trust center info (TC mode, TC EUI64 and TC link key)
Network management info (active channels, manager node ID and update ID)
Parent info (parent node ID and parent EUI64)
Child table (nodeID, EUI64 and information flags) – This is only duplicated for nodes that are coordinators on both participating networks.

Tokens that are not network-related are stored only once per node. These tokens return the same value regardless of the application network index context.

6. Multi-Network Timing Considerations

6.1 Leaving the Always-On Network

If the node participates in an always-on network as coordinator or router, it is important that the application does not poll and/or send data too frequently on the sleepy network(s). Every poll on the sleepy network(s) results in a temporary absence from the always-on network, which directly affects the throughput of the always-on network. This section provides the results of experimental measurements performed on Silicon Labs devices. This information will help the application designer to avoid throughput degradation on the always-on network. As we will show later in this section, a certain threshold in terms of “away time” from the always-on network should not be exceeded in order to maintain the throughput on the always-on network at an acceptable level.

The following table provides the average time a multi-network node spends during a complete network switch, and of typical polling and data-sending transactions of a sleepy end device. Data packets exchanged during the tests determining average time were frames 127 bytes long, or the highest size allowed by the 802.15.4 physical layer.

Table 6.1. Polling Sequences Average Time

Event sequence	Average time
Network switch	420 µs
POLL + NO DATA	2.26 ms
POLL + DATA	8.02 ms
DATA + POLL + NO DATA	8.82 ms
DATA + POLL + DATA	14.52 ms

A complete network switch, which involves retuning the radio on a different channel with different transmission power, takes about 420 µs. During the network switch, the node will not be able to receive or transmit on any network.

It takes on average about 2.25 ms for a sleepy end device to poll a parent that does not have data to transmit to the child, while it takes about 8 ms to poll the parent and receive a data packet from the parent.

Similarly, it takes about 8.8 ms to send data to the parent and then poll the parent, without receiving a data packet. Finally, it takes about 14.5 ms to send data to the parent, poll for data, and receive a data packet from the parent.

Note: For each polling transaction, add the network switch time twice to the overall transaction time (the first switch to the sleepy network and second switch to the always on network).

To estimate how the throughput of the always-on network degrades as the traffic on the sleepy network increases, we deployed a three-node network as shown in Figure 2.1 Implementation Examples on page 3, where the multi-network node is the coordinator of an HA network (Network A) and is joined as sleepy end device to an SE network (Network B).

Traffic was sent continuously on Network A so that maximum throughput is always achieved. Traffic on Network B was exchanged at different rates. All the data packets exchanged in these tests were encrypted at both network and APS layers and had an 82-byte payload (the maximum achievable payload with network and APS encryption for a single ZigBee fragment).

Figure 6.1 Throughput of the Always-On Network on page 10 illustrates how the maximum achievable throughput on Network A degrades as the traffic on Network B increases. All the values are expressed as a fraction of the maximum throughput achieved when no traffic is exchanged on Network B (about 53.5 packets per second).

The interval between SED network activity in the following figure indicates how often the multi-network node leaves the always-on network to perform a data transaction on the sleepy network. In an SED data transaction, the multi-network node polls, the coordinator sends a data packet, and the multi-network node sends an APS acknowledgment to the coordinator.

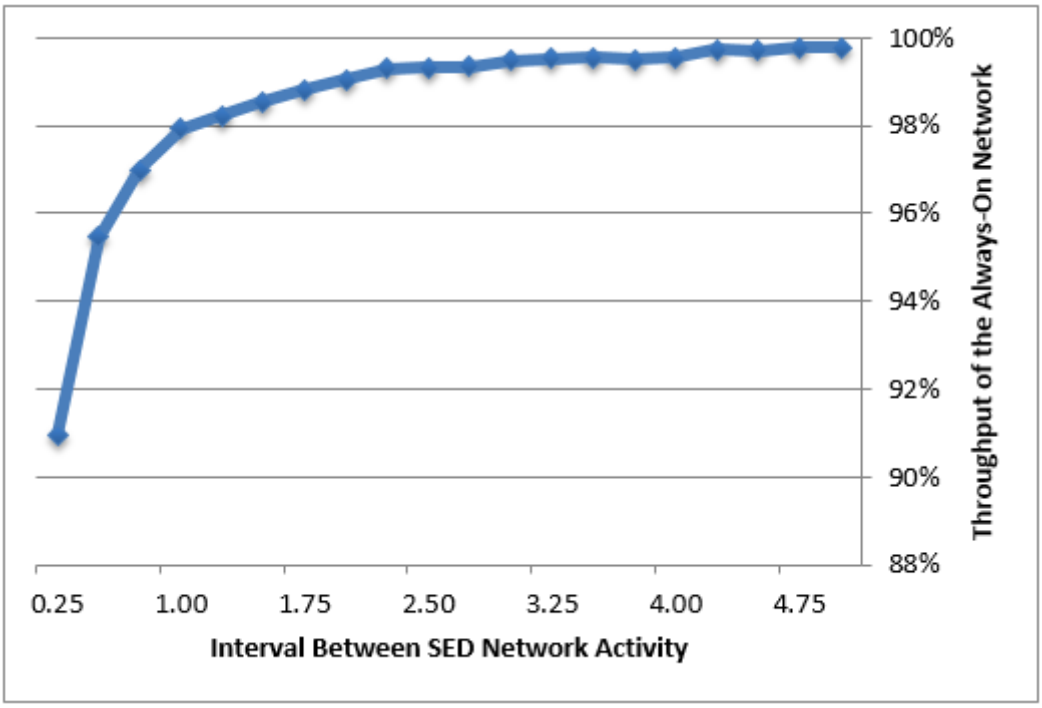

Figure 6.1. Throughput of the Always-On Network

We found the following results:

- By leaving the always-on network every 5 seconds, a multi-network node achieves a throughput of 99.8%,
- By leaving the always-on network every 2 seconds a multi-network node achieves a throughput of 99.0%,
- By leaving the always-on network every quarter of a second the throughput drops to about 91%.

Notice that these tests represent an average scenario where traffic is non-bursty, that is, every two subsequent data transactions on the sleepy network are well-spaced. Therefore, the multi-network node is always able to go back to the always-on network after one data transaction on the sleepy network. Other tests have shown that heavy bursts of outgoing traffic on the sleepy network can lead a multi-network node to spend longer time intervals on the sleepy network, which in turn can further reduce the throughput on the always-on network. For instance, exchanging the same amount of traffic as one data transaction every quarter of a second in a bursty fashion would further reduce the throughput to about 86.4%.

To summarize, traffic on the sleepy network directly affects the throughput of the always-on network. However, both the rate of such traffic and also its distribution in time are important. The application designer should take into account these results when defining the type and the amount of traffic that will be exchanged on the sleepy network.

Note: The application designer should keep the length of a single interval of time away from the always-on network as short as possible, by spacing polls and data sends on the sleepy network(s).

The multi-network application designer has full control of how long and how often a multi-network node leaves the always-on network for the sleepy network. A single network sleepy end device automatically polls again for data if the incoming packet from the parent has

the frame pending bit set. However, if a multi-network node is also participating in an always-on network, the automatic poll is delayed by 100 ms.

Note: A multi-network node that participates in an always-on network and a sleepy network is guaranteed to switch back to the always-on network after one poll on the sleepy network. If the frame pending bit of the incoming packet is set, the node will poll again after 100 ms.

Some special operations can occasionally occur on the sleepy network that can cause a multi-network node to stay on the sleepy network for a prolonged time interval. We measured how throughput on the always-on network is affected during these special operations. Please refer to the following table for more details.

Table 6.2. Throughput of the Always-On Network During Special Operations on the Sleepy Network

Special operation on the sleepy network	Average time	Throughput
Join (HA)	0.63s	95.9%
Find network + Join (HA)	1.81s	16.2%
Join + Registration (SE)	29.4s	94.7%
Find network + Join + registration (SE)	31.0s	88.7%

IoT Portfolio
www.silabs.com/products

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

www.silabs.com