AN724

DESIGNING FOR MULTIPLE NETWORKS ON A SINGLE ZIGBEE CHIP

(Formerly document 120-5077-000)

Beginning with version 4.7 of the EmberZNet PRO stack, Silicon Labs introduced a multi-network stack feature that allows a regular single-radio Ember node to be concurrently part of more than one distinct network. For EmberZNet PRO 4.7, multi-network support is limited to two networks. More than two networks will be supported in the future.

A new library named “multi-network-library” provides the multi-network functionality. A stub version of the library termed “multi-network-stub-library” can be used to save some flash in case the multi-network feature is not required.

Multi-networking is achieved by timesharing the radio on the networks. The networks can live on the same channel or different channels. The networks can operate using different security settings, different transmission power, different network parameters such as short ID, PAN ID, extended PAN ID, node role, and so on. The only parameter that stays the same on all networks is the EUI64 of the node.

Some limitations and restrictions are enforced by the multi-network stack and should be taken into account during the design of a multi-network application. These limitations are mostly related to the role the node assumes on the networks and are discussed in detail in the remainder of this document.

New in This Revision

Document renumbering.

Contents

1 Node Roles

With the multi-network stack, a node can be concurrently active on more than one network. In Figure 1 we show a two-network topology where a multi-network node participates in both networks. On network A the multi-network node acts as coordinator for node 1, which is a regular end device. On network B the multi-network node behaves as a sleepy end device and is joined to node 2, which is a single-network coordinator.

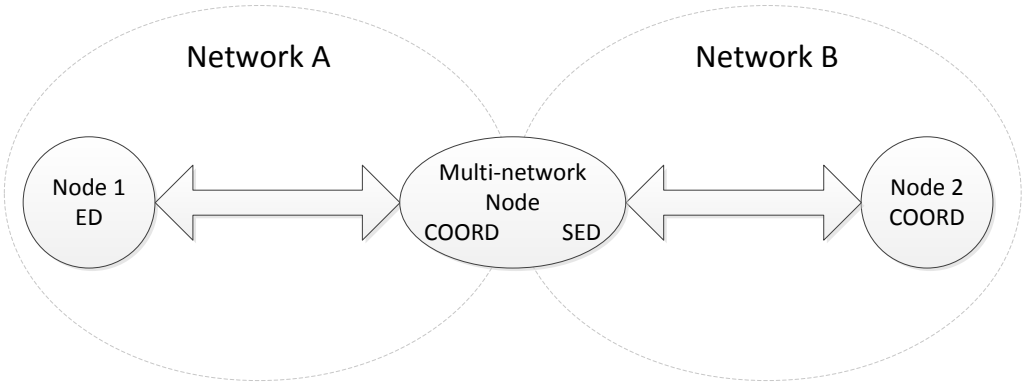

Figure 1. Two-Network Implementation Example

The stack enforces some restrictions regarding the role a multi-network node can assume on the networks. One such restriction applies to always-on networks.

An "**always-on**" network is a network on which a node participates as coordinator, router or (non-sleepy) end device. Since every node has only one radio, it can only listen to one network at a time. Non-sleepy nodes are expected to be always active by the neighboring nodes, therefore a multi-network node should spend most of its time on the always-on network.

Note: A multi-network node can participate in at most one always-on network.

Obviously, a node can also act on all networks as sleepy end device. In this case, it can sleep when it is not active on any network. Table 1 shows all the legal role combinations of a multi-network node on two networks. A node can assume any role on one network, while it **must** participate as a sleepy end device on the other network(s). The order in this table is not relevant. In other words, the node could be a sleepy end device on Network A and any other role on Network B.

Table 1. Allowed Role Combinations

Network A	Network B
Coordinator (always on)	Sleepy end device
Router (always on)	Sleepy end device
End device (always on)	Sleepy end device
Sleepy end device	Sleepy end device

Multi-networking is achieved by timesharing the radio between the networks. The Ember network scheduling algorithm seamlessly takes care of switching among 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(s) is completed, the node immediately switches back to the always-on network until a new polling or data sending transaction is initiated.

Note: A node that is a coordinator/router on one network and a sleepy end device on the others won't 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 saving energy.

2 Network Context and Stack APIs

The Ember multi-network 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 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. Table 2 lists the APIs provided to set and get the application current network.

Table 2. Multi-Network Stack APIs

API	Description
`int8u emberGetCurrentNetwork()`	Returns the index of the current application network context.
`EmberStatus emberSetCurrentNetwork(int8u index)`	Sets the current application current network context.
`int8u 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 getter and setter functions for the application network context. The application network context is set by passing the network context index, that is, either 0 or 1. If any other value to is passed to `emberSetCurrentNetwork(int8u index)` the API returns the error code `EMBER_INVALID_CALL`.

Notice that no restriction is placed on the role of a node on a specific network context. In other words, a node can form or join an always-on network either on network 0 or network 1. Below is a code example of a node forming on network 1 and joining as a sleepy end device on network 0.

```
emberSetNetwork(1);
emberFormNetwork(params);
emberSetNetwork(0);
emberJoinNetwork(SLEEPY, params);
```

The `emberGetCallbackNetwork()` API makes the application aware of the network context a callback is referred to. For instance, when the stack calls `emberIncomingMessageHandler()`, the application inside the callback can call `emberGetCallbackNetwork()` to retrieve the index of the network to which the packet is related. The following is an example.

```
emberIncomingMessageHandler(  EmberIncomingMessageType type,
                              EmberApsFrame *apsFrame,
                              EmberMessageBuffer message)
{
      int8u nwkIndex = emberGetCallBackNetwork();

      // Processing the incoming packet coming from the network with
      // index nwkIndex
}
```

3 Application Framework Multi-Network Support

The Ember application framework provides multi-network support. We strongly encourage the application designer to use the application framework when developing a multi-network application. The application framework provides many advantages to the application developer in terms of reduced complexity, mostly related to how the framework seamlessly manages the different network contexts. The application framework takes care of switching the current network context so that all the API calls refer to the appropriate network. See UG102, *Application Framework Developer Guide*, for details on implementing multi-network functionality with the application framework.

In the following, we discuss in detail how the application framework handles different network contexts.

- **Endpoint mapping:** The Ember application framework maintains a mapping between network contexts and disjoint sets of endpoints. For example, network 0 is associated to the set of endpoints {0,1,2,3} while network 1 is associated to 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:** The Ember application framework offers the application designer many callbacks. It ensures that the network context is always correctly set when an application framework callback is invoked. For example, when `emberAfPreMessageReceivedCallback()` is called, the current network context gets set to that of the receiving network.

The application framework provides a set of APIs for dealing with network context, summarized in Table 3. 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 these APIs.

Table 3. Multi-Network Application Framework APIs

API	Description
`EmberStatus emberAfPushNetworkIndex(int 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()`	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()`.

API	Description
`EmberStatus emberAfPushEndpointNetworkIndex( int8u 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()`.
`int8u emberAfPopNetworkIndex()`	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 `emberAfPushEndPointtNetworkIndex()`.
`EmberStatus emberAfPrimaryEndpointForNetworkIndex(int8u index)`	Returns the primary endpoint of the given network index or 0xFF if no endpoints belong to the network.
`int8u emberAfPrimaryEndpointForCurrentNetworkIndex()`	Returns the primary endpoint of the current network index or 0xFF if no endpoints belong to the current network.
`int8u emberAfNetworkIndexFromEndpoint(int8u endpoint)`	Returns the network index of a given endpoint or 0xFF if the endpoint does not exist.
`int8u emberAfNetworkIndexFromEndpointIndex(int8u 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.

4 Tokens

Each network has its own set of network-related tokens. Tokens can be accessed using the usual APIs by simply changing the application network context before invoking the API. The tokens that have been made “per network” are summarized in Table 4:

Table 4. Multi-Network Tokens

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)

Only one instance of each of the non-related network tokens is maintained. Therefore, regardless of the application network context, token-accessing APIs always refer the same (unique) instance of a non-related network token.

5 Leaving the Always-On Network

A multi-network node can act either as a sleepy end device on all networks or can be a coordinator/router/end device on one, always-on, network, and a sleepy end device on the other networks.

If the node participates in the 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 Ember 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.

Table 5 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 5. 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 won’t 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).

In order 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 1, 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).

In Figure 2 we show 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 Figure 2 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.

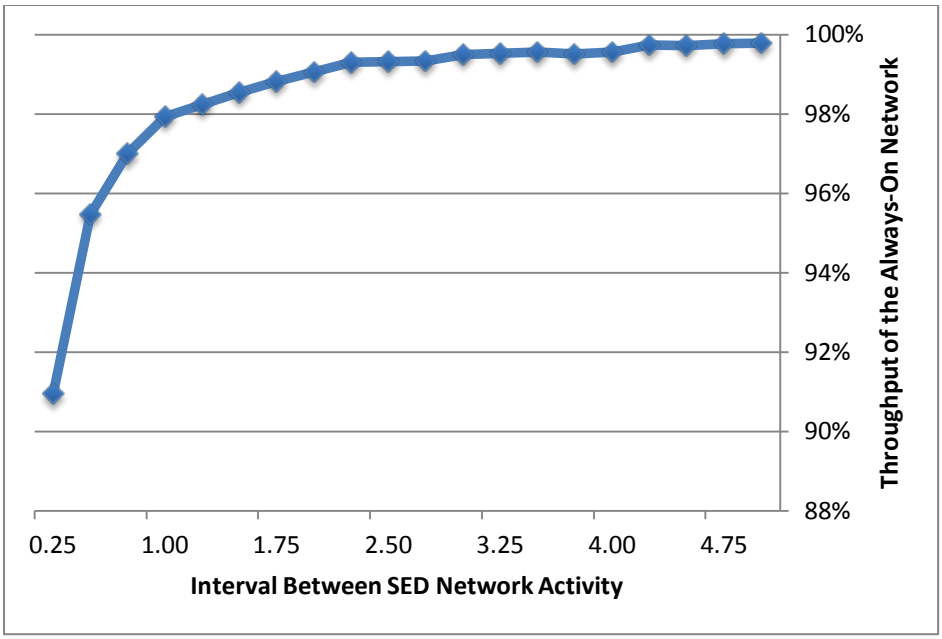

Figure 2. Throughput of the Always-On Network

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 showed that heavy bursty 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 Table 6 for more details.

Table 6. 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%

NOTES:

CONTACT INFORMATION

Silicon Laboratories Inc.

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

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