



SmartMesh IP Mote User Guide

Advance Information

This document contains advance information of a product in development. All specifications are subject to change without notice. Consult LTC factory before using.




Table of Contents

| | | |
|-----|----------------------------|----|
| 1 | About This Guide | 5 |
| 1.1 | Related Documents | 5 |
| 1.2 | Conventions Used | 5 |
| 1.3 | Revision History | 6 |
| 2 | Mote State Machine | 7 |
| 3 | Joining | 10 |
| 3.1 | Manual Join | 10 |
| 3.2 | Auto join | 11 |
| 3.3 | Joining adjacent network | 11 |
| 4 | Services | 12 |
| 4.1 | Requesting bandwidth | 12 |
| 4.2 | Back-off mechanism | 13 |
| 5 | Communication | 14 |
| 5.1 | Sockets | 14 |
| 5.2 | UDP Port Assignment | 14 |
| 5.3 | Sending and Receiving Data | 15 |
| 6 | Events and Alarms | 16 |
| 7 | Factory Default Settings | 17 |
| 8 | Power Considerations | 18 |
| 8.1 | Power Source Information | 18 |
| 8.2 | Routing Mode | 18 |
| 8.3 | Join duty cycle | 19 |



Trademarks

Eterna, Mote-on-Chip, and SmartMesh IP, are trademarks of Dust Networks, Inc. The Dust Networks logo, Dust, Dust Networks, and SmartMesh are registered trademarks of Dust Networks, Inc. LT, LTC, LTM and  are registered trademarks of Linear Technology Corp. All third-party brand and product names are the trademarks of their respective owners and are used solely for informational purposes.

Copyright

This documentation is protected by United States and international copyright and other intellectual and industrial property laws. It is solely owned by Dust Networks, Inc. and its licensors and is distributed under a restrictive license. This product, or any portion thereof, may not be used, copied, modified, reverse assembled, reverse compiled, reverse engineered, distributed, or redistributed in any form by any means without the prior written authorization of Dust Networks, Inc.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g) (2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015 (b)(6/95) and DFAR 227.7202-3(a), and any and all similar and successor legislation and regulation.

Disclaimer

This documentation is provided “as is” without warranty of any kind, either expressed or implied, including but not limited to, the implied warranties of merchantability or fitness for a particular purpose.

This documentation might include technical inaccuracies or other errors. Corrections and improvements might be incorporated in new versions of the documentation.

Dust Networks does not assume any liability arising out of the application or use of any products or services and specifically disclaims any and all liability, including without limitation consequential or incidental damages.

Dust Networks products are not designed for use in life support appliances, devices, or other systems where malfunction can reasonably be expected to result in significant personal injury to the user, or as a critical component in any life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness. Dust Networks customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify and hold Dust Networks and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Dust Networks was negligent regarding the design or manufacture of its products.

Dust Networks reserves the right to make corrections, modifications, enhancements, improvements, and other changes to its products or services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to Dust Network’s terms and conditions of sale supplied at the time of order acknowledgment or sale.



Dust Networks does not warrant or represent that any license, either express or implied, is granted under any Dust Networks patent right, copyright, mask work right, or other Dust Networks intellectual property right relating to any combination, machine, or process in which Dust Networks products or services are used. Information published by Dust Networks regarding third-party products or services does not constitute a license from Dust Networks to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from Dust Networks under the patents or other intellectual property of Dust Networks.

Dust Networks, Inc is a wholly owned subsidiary of Linear Technology Corporation.

© Dust Networks, Inc. 2012 All Rights Reserved.



1 About This Guide

1.1 Related Documents

The following documents are available for the SmartMesh-enabled network:

- SmartMesh IP Quick Start Guide
- SmartMesh IP Network User Guide
- SmartMesh IP Manager User Guide
- SmartMesh IP Manager CLI Guide
- SmartMesh IP Manager API Guide
- [SmartMesh IP Mote User Guide](#)
- SmartMesh IP Mote API Guide
- SmartMesh IP Mote CLI Guide

1.2 Conventions Used

The following conventions are used in this document:

`Computer type` indicates information that you enter, such as specifying a URL.

Bold type indicates buttons, fields, and menu commands.

Italic type is used to introduce a new term



Tips provide useful information about the product.



Informational text provides additional information for background and context



Notes provide more detailed information about concepts.



Warning! Warnings advise you about actions that may cause loss of data, physical harm to the hardware or your person.



code blocks display examples of code or API

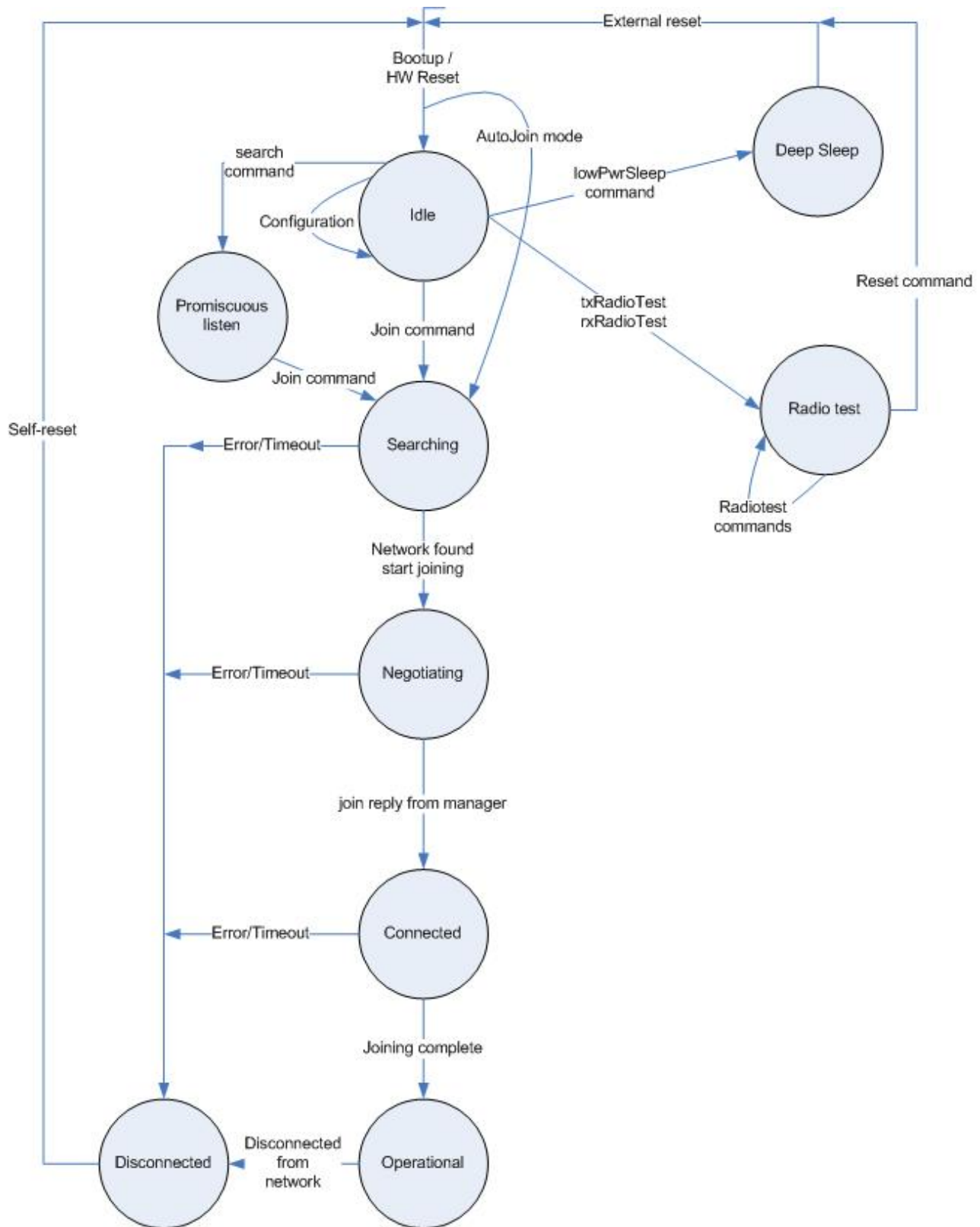
1.3 Revision History

| Revision | Date | Description |
|----------|------------|-----------------|
| 1 | 07/17/2012 | Initial release |



2 Mote State Machine

The following state machine describes the general behavior of a mote during operation:



The mote states are as follows:



- **Idle** - While in this state, the mote accepts configuration commands. This state is skipped if the mote is configured to auto-join.
- **Deep Sleep** - The mote enters Deep Sleep when it receives the *lowPowerSleep* command from the attached serial processor. In this state, the device can no longer respond to serial commands and must be reset to resume normal operation. For power consumption information, refer to the mote product datasheet.
- **Promiscuous Listen** - A special search state, invoked by the *search* command, where the mote listens for advertisements from any network ID, and reports heard advertisements. The mote will not attempt to join any network, and will proceed to the **Searching** state when given the join command.
- **Searching** (unsynchronized search) - The mote is searching for the network that matches its network id. It keeps its radio receiver on with a configurable join duty cycle.
- **Synchronized Search (not shown)** - Short period at the end of searching. The device has heard an advertisement and has synchronized to the network. It keeps its radio receiver on at the configured join duty cycle, listening for additional potential neighbors.
- **Negotiating** - The device started joining the network
- **Connected** - The device heard join reply from the manager and is being configured by it.
- **Operational** - The device finished network joining and is ready to send data.
- **Disconnected** - The device is disconnected from the network.
- **Radio Test** - In this state the device is executing Radio Test commands. It must be reset to return to normal operation.



3 Joining

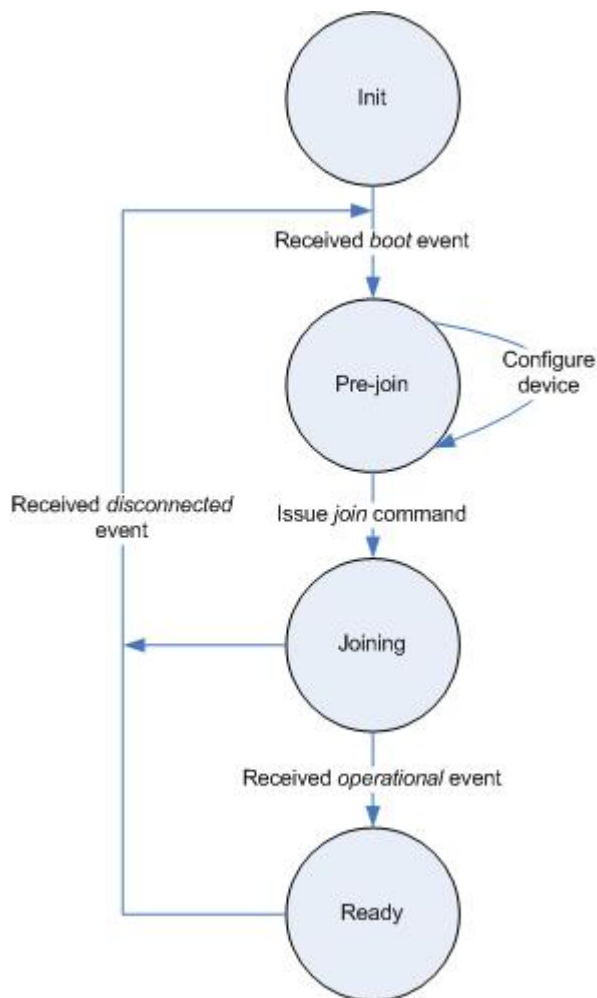
3.1 Manual Join

To join the network, the device must be configured with the following parameters that bind the device to the network.

- *networkId*
- *joinKey*

These parameters are persistent and may be set once during commissioning. In addition, other parameters such as *joinDutyCycle*, *routingMode*, *powerSourceInfo* all affect joining behavior and can be optionally set.

The client application should use the following state machine to join the network:





In this diagram, the following states are assumed:

- **Init** - in this state the application is booting up and initializing. The mote may be held in hardware reset until the app is ready to communicate with it.
- **Pre-join** - Once the mote boots up (as indicated by the *boot* event), the application may proceed to configure it by making a number of *setParam* API calls. At the end of this state, the application may issue a *join* command.
- **Joining** - In this state the mote is joining the network. Successful join will eventually be reported via the *Operational* event notification. A failed join will be reported via the *joinFail* event notification.
- **Ready** - In this state the device is completed joining and is part of the network. The application may proceed to request bandwidth and communicate wirelessly.

3.2 Auto join

The mote may be configured to automatically search for and start joining its network after reboot. This setting is controlled via persistent *autoJoin* parameter. In this case, no explicit *join* command is required. Note that all parameters, such as network id, power source, etc. must be pre-configured.



If the device is configured to auto join, radio test functionality cannot be exercised.

3.3 Joining adjacent network

In some cases the network id may not be known in advance. To find out which networks are in the proximity of the device, one can use the *search* API command. This command puts the device into promiscuous listen mode. In this mode, all received advertisements are reported via *advReceived* notification. After the correct network id has been established, the user may set it via *setParam<networkId>*, and follow up with a *join* command.

The following is the summary of the steps to follow after the mote boots up:

1. Put the mote into promiscuous listen state (*search* command)
2. Process *advReceived* notifications
3. Configure network id (*setParam<networkId>*)
4. Start joining (*join* command)



4 Services

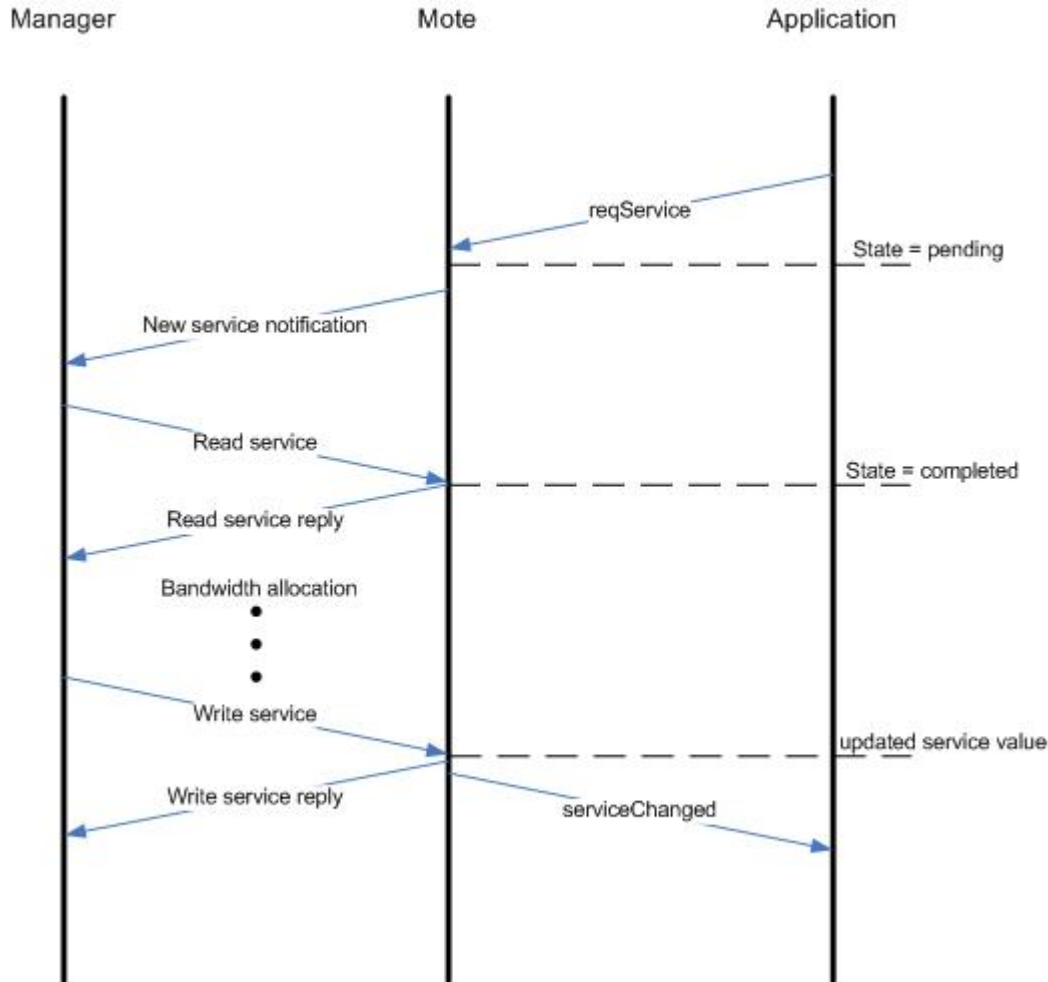
4.1 Requesting bandwidth

Once a mote is in the network and has reached Operational state, it can begin to send data packets. Initially all motes will receive a proscribed amount of upstream bandwidth (the global *base bandwidth*) to the manager - if this is not sufficient for mote's publishing rates, the application may request additional network bandwidth to a destination – this is called asking for a *service*. To find out the allocation to a destination, the application may use *getServiceInfo* API. Note that a mote will receive base bandwidth for communication with the manager even without explicitly asking for it.

A service is identified by its in-mesh destination and includes aggregate bandwidth needed by the mote. Only one service is allowed per destination, so an application that generates packets at different rates must request a single service equal to the total aggregate period. Currently the only in-mesh destination that motes can send traffic to is the manager (mesh address 0xFFFE).

When the application uses *requestService* API, the mote sends a notification to the manager about a pending service request. At this time the service state returned via API will be *pending*. Once the manager responds to the service request from the mote, the service state will change to *completed*. After getting the service request from the mote the manager stores most recent value requested and will notify the mote any time the bandwidth allocation changes – the app will receive a *serviceChanged* notification. Note that such notification may come at any point and any number of times if the network conditions change. The application may change its service requirements at any time using another *requestService* – the manager will always treat the last request as the most up-to-date.

The following transaction diagram demonstrates what is occurring between the application, mote and the manager during service request:



4.2 Back-off mechanism

The application can begin sending data immediately via the `sendTo` API after requesting a service, however it is required to back off, such that at any point it will only publish at a level that the network can tolerate. If multiple variables are being published to the same destination, the app should aggregate the total services required and make a single request.

The following back off algorithm is recommended:

- If `RC_NO_RESOURCES` is received, double the interval between packets. If held off again, then it goes to 3x, then 4x, ..., 255x.
- If packet transmission has been held off and is now succeeding, the interval decreases along the same pattern, 5x, 4x, ..., 1x, as the queue continues to have space.



5 Communication

5.1 Sockets

A socket is an endpoint of communication flow between a mote and another IP device. Mote sockets are loosely based on the Berkeley sockets standard. In order to communicate, the application must open a socket and bind it to a port. An application that terminates multiple ports may open multiple sockets.

Here's a normal sequence of using a socket:

1. Call *openSocket*
2. Call *bindSocket*
3. Use *sendTo* and/or process *receive* notifications. Repeated calls to *sendTo* can be made on the open socket.
4. Once done with the socket, call *closeSocket*

Currently SmartMesh IP only supports UDP sockets.

5.2 UDP Port Assignment

UDP ports in the range of 0xF0B0-0xF0BF are most efficiently compressed inside the mesh, and should be used whenever possible to maximize useable payload.

On the mote, the following port assignment is used:

| Port | Description |
|---------------|---|
| 0xF0B0 | Management traffic between manager and mote |
| 0xF0B1 | Reserved (used by OTAP) |
| 0xF0B2-0xF0B7 | Reserved |
| 0xF0B8-0xF0BF | Available for application |

On the manager, the following port assignment is used:

| Port | Description |
|--------|---|
| 0xF0B0 | Management traffic between manager and mote |
| 0xF0B1 | Reserved (used by OTAP) |



| | |
|---------------|---------------------------|
| 0xF0B2 | Reserved |
| 0xF0B3-0xF0B7 | Reserved |
| 0xF0B8-0xF0BF | Available for application |

Other ports may be used at a penalty of 3 bytes of payload.

5.3 Sending and Receiving Data

Once a socket is created, the application may send data to any IPv6 device using the `sendTo` API, including the manager. The manager's IPv6 address is `FF02::02`. If a packet is sent to the manager, it will be turned into a Data notification on manager's Serial API. A packet sent to any other IPv6 address will be turned into an IP Data Notification on the manager's Serial API.

Wireless data that the application sends is highly reliable (typically better than 99.9%), but end-to-end delivery is not guaranteed with UDP. If the application cannot tolerate any lost packets then application layer reliable messaging must be provided by the customer's application.

The application may only receive packets on ports for which sockets are open and bound to a particular port number. Once this is done, the mote will forward all packets received on that port the application using the receive notification.



6 Events and Alarms

The mote API includes events and alarms that allow an application to have better visibility of mote states and conditions. An alarm is an ongoing condition, such as an error in non-volatile memory or a low buffer condition. To read current alarms, the application can use `getParameter<moteStatus>` API.

By contrast, an event is defined as a discrete occurrence in mote or network operation. Examples of events include a mote startup event, or a change in alarm condition. The application can control which events it is subscribed to by using `setParam<eventMask>` API call.



7 Factory Default Settings

The mote ships with the following factory defaults:

| Parameter | Default value |
|--------------------------|---|
| Network Id | 1229 |
| Transmit Power | +8 dBm |
| Join Key (16 bytes, Hex) | 44 55 53 54 4E 45 54 57 4F 52 4B 53 52 4F 43 4B |
| OTAP Lockout | 0 (permitted) |
| Routing Mode | 0 (enabled) |
| Join Duty Cycle | 64 (25%) |
| maxStCurrent | 0xFFFF (no limit) |
| minLifetime | 0 (no limit) |
| currentLimit | 0 (none) |
| Auto Join | off |



8 Power Considerations

8.1 Power Source Information

For full description, refer to the documentation of `setParam<pwrSrcInfo>` API.

Of the parameters described in the *pwrSrcInfo* documentation, only `maxStCurrent` is used to make decisions by the SmartMesh IP manager. In assigning links, the manager will assume that each RX and TX link receives a maximum-length packet. Additional links are assigned to the mote only until `maxStCurrent` is met. Note that there is a minimum number of links required for operation in the network and that setting a `maxStCurrent` below this threshold will result in the mote getting the minimum link configuration, effectively ignoring `maxStCurrent`. This threshold varies depending on the downstream frame multiplier and the randomly chosen base frame size, but is ~30 uA.

This single parameter also has an impact on backbone activity. For an upstream backbone, only "powered" motes with no current restriction, i.e motes with `maxStCurrent=0xFFFF`, are eligible to have upstream RX links in the backbone frame. Having powered motes is the only way to construct a multi-hop upstream backbone. For a bi-directional backbone, all motes have a RX link every two slots as all motes need to participate in listening on the backbone. This will happen regardless the power setting.

The other power parameters are not currently used by the manager but products should be designed to fill them in properly so that they work with future implementations. The `minLifetime` parameter will be used as a complement to `maxStCurrent`, and the manager will obey whichever limit is more strict. For example, if `minLifetime` results in a mote being able to have 10 links/s but `maxStCurrent` sets the limit at 12 links/s, then the 10 links/s value will be used. This parameter is most useful in networks wherein devices have different types of batteries. Additionally, we provide three sets of temporary current limits. The first set consists of `currentLimit_0`, `dischargePeriod_0`, and `rechargePeriod_0`. The intent here is to have `currentLimit_0` be higher than `maxStCurrent`. For example, a device that scavenges power may be able to source 40 uA of current throughout its lifetime but an on-board capacitor may be available to provide 100 uA of current for 1 minute at a time after which it needs 10 minutes to recharge. In this case, we would set `currentLimit_0 = 100`, `dischargePeriod_0 = 60` and `rechargePeriod_0 = 600`. By having three different such sets, a very general power profile can be described for each mote that allows for the enabling of different types of temporary services and bandwidth.

8.2 Routing Mode

Independent of the power setting, each mote is given a routing mode that may be changed via `setParam<routingMode>` API.



Setting `mode=non-routing` disables routing, meaning the *non-routing* mote will not be assigned children. This affords some real energy savings for the mote as it is not given advertisement links or a discovery RX link. These changes take effect even if the mote is set to `maxStCurrent=0xFFFF`. Note that setting all motes to non-routing forces the network into a 1-hop star topology with the AP as the only parent.

Setting `mote=routing` enables routing (default), meaning the mote could be assigned children. The mote will send advertisements and listen on discovery RX links for packets from other motes. It is not guaranteed that a mote with routing enabled will receive children as not all motes need children for an optimal network. A routing-enabled mote that happens to not have children is still called a *leaf*. A network where all motes are routing-enabled will have at least one leaf.

8.3 Join duty cycle

The `joinDutyCycle` parameter allows the microprocessor to control the join duty cycle - the ratio of active listen time to doze time (a low-power radio state) during the period when the mote is searching for the network. The [default](#) duty cycle enables the mote to join the network at a reasonable rate without using excessive battery power. If you desire a faster join time at the risk of higher power consumption, use the `setParameter<joinDutyCycle>` command to increase the join duty cycle up to 100%. Note that the `setParameter<joinDutyCycle>` command is not persistent and affects only the next join. For power consumption information, refer to the mote product datasheet. This command may be issued multiple times during the joining process. This command is only effective when the mote is in the IDLE and SEARCHING states.